Software and Systems Modeling
https://doi.org/10.1007/s10270-022-01040-x

SPECIAL SECTION PAPER

Composable partial multiparty session types for open systems

Claude Stolze[1] · Marino Miculan[1] · Pietro Di Gianantonio[1]

Received: 31 March 2022 / Revised: 19 July 2022 / Accepted: 6 September 2022

Abstract
Session types are a well-established framework for the specification of interactions between components of a distributed systems. An important issue is how to determine the type for an *open* system, i.e., obtained by assembling subcomponents, some of which could be missing. To this end, we introduce *partial sessions* and *partial (multiparty) session types*. Partial sessions can be *composed*, and the type of the resulting system is derived from those of its components without knowing any suitable global type nor the types of missing parts. To deal with this incomplete information, partial session types represent the *subjective* views of the interactions from participants' perspectives; when sessions are composed, different partial views can be *merged* if compatible, yielding a unified view of the session. Incompatible types, due to, e.g., miscommunications or deadlocks, are detected at the merging phase. In fact, in this theory the distinction between global and local types vanishes. We apply these types to a process calculus for which we prove *subject reduction* and *progress*, so that well-typed systems never violate the prescribed constraints. In particular, we introduce a generalization of the progress property, in order to accommodate the case when a partial session cannot progress not due to a deadlock, but because some participants are still missing. Therefore, partial session types support the development of systems by incremental assembling of components.

Keywords Multiparty session types · Process algebras · Open systems

1 Introduction

The design and implementation of distributed applications is notoriously difficult and error-prone. In order to tame the complexity of this task, we look for *compositional* methods, which allow us to examine processes and subsystems in isolation and formalize their interactions. *Multiparty session types* (MPST) are a well-established theoretical and practical framework for the specification of the interactions between components of a distributed systems [11,13,16–18,29]. The gist of this approach is to first describe the system's overall behavior by means of a *global type*, from which a local specification (*local type*) for each component can be derived. The system will behave according to the global type if each component respects its local type, which can be ensured by means of, e.g., static type checking [19,27]. The global type is given beforehand by the programmer, and the type system checks that the local behavior of all the participants, given by *local types* conform to it. Therefore, session types support a *top-down* style of coding: first the designer specifies the behavior from a global perspective, then the programmers are given the specifications for their modules. On the other hand, these session types do not fit well *bottom-up* programming models, where systems are built incrementally by composing existing reusable components, possibly with dynamic bindings. Some examples are (micro)service-oriented and component-based architectures, and *containers* [24]. In these situations, components could offer "contracts" in the form of, e.g., session types; then, when these components are connected together, we would like to derive the contract for the resulting system from components' ones. The system becomes a new component which can be used in other assemblies, and so on.

To this end, we need to infer the type for an *open* system (i.e., where some parts may be still missing) using the types of the known components, in a compositional way and without knowing any global type. This is challenging. As an example,

Communicated by Gwen Salaun.

✉ Marino Miculan
marino.miculan@uniud.it

Claude Stolze
claude.stolze@uniud.it

Pietro Di Gianantonio
pietro.digianantonio@uniud.it

[1] Department of Mathematics, Computer Science and Physics, University of Udine, Udine, Italy

let us consider a protocol from [29] with three participants: a server s, an authorization server a and some client c. First, s sends to c either a request to login, or to cancel. In the first case, c sends a password to a, and a sends a Boolean to s (telling whether c is authorized). In the second case, c tells a to quit. Using the syntax of [29], the two server processes have the following types:

$$S_s := c\oplus\{\text{login}.a\&\text{auth}(\text{Bool}), \text{cancel}\}$$
$$S_a := c\&\{\text{passwd}(\text{Str}).s\oplus\text{auth}(\text{Bool}), \text{quit}\}$$

Let us suppose that we have implementations a and s for S_a, S_s. To prevent miscommunications, we would like to verify that these two processes work well together, e.g., we have to ensure that a can send the message auth(Bool) to s iff s is waiting for it. This corresponds to see these two processes as a single system $a|s$, and to check that $a|s$ is well-typed *without knowing the behavior of clients*; more precisely, we have to figure out a session type for $a|s$ from S_a and S_s. This is difficult, because the link that propagates the choice made by s to a is the missing client c, so we have to "guess" its role without knowing it.

In this paper, we address this problem by introducing *partial sessions* and *partial (multiparty) session types*. Partial session types generalize global types with the possibility to type also partial (or *open*) sessions, i.e., where some participant may be missing. The key difference is that while a global type is a complete, "platonistic" description of the protocol, partial session types represent the *subjective* views from participants' perspectives. We can *merge* two sessions with the same name but from two different "point of views," whenever their types are *compatible*; in this case, we can compute the new, unified, session type from those of the components. In this way, we can guarantee important properties (e.g., absence of deadlocks) about partial session without knowing all participants beforehand, and without a complete global type. In fact, the distinction between local and global types vanishes: local types correspond to partial session types for sessions with a single participant, and global types correspond to *finalized* partial session types, i.e., in which no participant is missing.

Defining "compatibility" and how to merge partial session types is technically challenging. Intuitively, the semantics of a partial session type is the set of all possible execution traces (which depend on internal and external choices). We provide a merging algorithm computing a type covering all the possible synchronizations of these traces. Incompatible types, due to, e.g., miscommunications or deadlocks, are detected when no synchronization is possible. Also the notion of *progress* has to be revisited, to accommodate the case when a partial session cannot progress not due to a deadlock, but due to some missing participant.

Outline. The paper is organized as follows. We review some related work in Sect. 2. In Sect. 3, we introduce a formal calculus for processes communicating over multiparty sessions. Our theory of partial session types is presented in Sect. 4, and the type system for the process calculus is in Sect. 5. Central to this type system is the merging algorithm, which we describe in Sect. 6. A trace semantics of session types is given in Sect. 7, for which we show that the merging operation is sound and complete. The results about the semantics of session types will be useful also in Sect. 8, where we prove the crucial properties of subject reduction and progress for the process calculus under investigation. Finally, conclusions and directions for further work are in Sect. 9.

2 Related work

The present paper is a revised and significantly extended version of [30].

The problem of composing session types has been faced in several related work. Compositional choreographies are discussed in [26], with the same motivations as ours, but from a different perspective. The authors manage to compose choreographies using global types, but the global type of shared channels has to be the same. This is in contrast with our approach, where the processes may have different session types that we merge during the composition. Moreover, their typing judgments use the sets of all the participants (there called *roles*); more precisely, types for channels keep track of the "active" role, the set of all roles in the global type, and the roles actually implemented by the choreography under typing. On the other hand, we do not need to specify neither the complete set of participants nor the "active" role.

Synthesis of choreography from local types has been studied also in [22], but with no notion of "partial types" and no distinction between internal/external choice. Graphical representations of choreographies (as communicating finite-state machines) and global types have been used in [23], where an algorithm for constructing a global graph from asynchronous interactions is given.

An interesting approach for connecting systems via some intermediating agent has been investigated in [2–6]. Two independent global types G_1, G_2 with different participants can be composed through participant h in G_1 and k in G_2 where h and k relay the message they receive to each other. In particular, in [5] behaviors of systems are represented by means of Communicating Finite State Machines (CFSMs); these can be connected along compatible interfaces by means of suitable *gateway* CFSMs, which can be effectively constructed.

Finally, Scalas and Yoshida [29] do not use global types altogether: behaviors of systems are represented by sets of local types, over which no consistency conditions are required, and behavioral properties can be verified using model checking techniques. The approach of [29] can model situations which are not captured by our partial session types, at least because it supports recursion; on the other hand, our approach allows for incremental, compositional verification, whereas [29] checks the correctness of sessions only after all participants' specifications are available.

A problem similar to ours is considered in [9], where the authors introduce a type system for the Conversation Calculus, a model for service-oriented computing. Conversation types of parallel processes can be merged like in our approach, but the underlying computational model is quite different.

Semantics of concurrent processes can be given using Mazurkiewicz trace languages [28]. Semantics can also be defined using event structures, as in [12], where they are used for defining equivalent semantics for processes and their global types. Interestingly, the semantics for global types proposed in [12] is similar to the representation of Mazurkiewicz trace languages as event structures given in [28]. Mazurkiewicz trace languages have also been used to characterize testing preorders on multiparty scenarios [14]. A denotational semantics based on Brzozowski derivatives that corresponds to bisimilarity is given in [21].

Another semantics of processes (but for binary session types) which records exchanged informations is given in [1]. This semantics is similar to the relation-based model of linear logic [7] and is not based on traces. It would be interesting to investigate if this alternative semantics can be extended to MPST and how the merging operation would be interpreted in it. The relationship between category theory and session types has also been investigated in [20,31].

3 A calculus for processes over multiparty sessions

Our language for processes is inspired by [11], which is in turn inspired by [32]; as in those works, we consider *synchronous* communications.

Simple data language Our process algebra is parametrized by an auxiliary language for simple data, that we will call $\mathfrak{D}$. We suppose $\mathfrak{D}$ to be a typed language whose types $A, B, \ldots$ are taken from some set $\mathbb{B}$. The set of terms for $\mathfrak{D}$ are ranged over by $M, N, \ldots$, and contains variables $x, y, \ldots$ and values $v_1, v_2, \ldots$ In particular, we assume there is a type Bool with two values tt, ff. As usual, a *(term typing) environment* E is a map from a finite set of variables to types; then, we assume to have a (term) typing judgment $E \vdash_{\mathfrak{D}} M : A$. We will omit the subscript $_{\mathfrak{D}}$ when clear from the context.

We assume also a normalization relation $M \downarrow v$ between terms and values; it can be partial or non-deterministic, but we assume that $\mathfrak{D}$ is normalizing for Booleans, i.e., for M

$$
\begin{aligned}
P, Q, R ::= {} & \overline{x}^{p\tilde{q}}\langle M\rangle.P \mid x^{pq}(y : A).P \\
& \mid \overline{x}^{p\tilde{q}} \triangleright l.P \mid x^{pq} \triangleleft \{l_1 : P_1, \ldots, l_n : P_n\} \\
& \mid \overline{x}^{p\tilde{q}}(y).P \mid x^{pq}(y).(P \parallel Q) \\
& \mid \mathrm{close}(x) \mid \mathrm{wait}(x).P \mid (P \mid_x Q) \\
& \mid (\nu x)P \mid \text{if } M \text{ then } P \text{ else } Q
\end{aligned}
$$

Fig. 1 Syntax of the process calculus for multiparty session

such that $\varnothing \vdash_{\mathfrak{D}} M$: Bool, then $M \downarrow tt$ or $M \downarrow ff$ (or both). This will be important for proving the progress property of our process calculus.

Syntax of processes Let us note $p, q, p_1, p', \ldots$ for *participant names*, taken from some set $\mathfrak{P}$, and $\tilde{p}$ for a finite non-empty set of participants $\{p_1, \ldots, p_n\}$. Given a simple data language $\mathfrak{D}$, the syntax of *processes for multiparty sessions* is provided in Fig. 1.

We provide a brief description of these constructors.

- The process $\overline{x}^{p\tilde{q}}\langle M\rangle.P$ executes a *synchronous send* of term M (belonging to $\mathfrak{D}$), as participant p, to all participants $\tilde{q}$ in session x; dually, $x^{pq}(y : A).P$ is a participant p waiting for an input (of type A) from q in session x, before continuing as P where the term variable y is replaced by the received term.
- The process $\overline{x}^{p\tilde{q}} \triangleright l.P$ sends label l in session x, as participant p, to $\tilde{q}$, and proceeds as P. For each participant $q \in \tilde{q}$, this label is received by process of the form $x^{qp} \triangleleft \{l_1 : P_1, \ldots, l_n : P_n\}$, which then proceeds as P_i if $l = l_i$.
- The process $\overline{x}^{p\tilde{q}}(y).P$ creates a fresh subsession handler y, sends it to $\tilde{q}$, and proceeds as P. This handler is received by processes of the form $x^{qp}(y).(Q \parallel R)$ (for each $q \in \tilde{q}$) which forks a process Q dedicated to the new session y, in parallel with the continuation R (on the previous session x).[1]
- Parallel composition of processes P and Q through session x is denoted by $P\big|_x Q$. A participant executes $\mathrm{close}(x)$ to end its communications on session x; after $\mathrm{close}(x)$ no further actions are possible. Thus, $\mathrm{close}(x)$ is for $\big|_x$ what 0 is for $\big|$ in CCS and the π-calculus. On the other hand, $\mathrm{wait}(x).P$ waits until all other participants on session x are gone, then it closes the session x and continues as P.
- Finally, $(\nu x)P$ is the usual restriction of session names, *à la* π-calculus, and "if M then P else Q" is the standard branching (i.e., "internal" choice).

[1] From a computational point of view, this "parallel input" corresponds to the programming practice to selectively share sessions between processes. This constructor allows us to enforce a discipline on the shared sessions in order to avoid deadlocks between processes. Moreover, it is motivated by connections with linear logic [11,32].

Notice that $\mathfrak{D}$-terms are only used as data that is sent or received, and for if-then-else clauses. The session name y is bound in processes of the form $(\nu y)P$, $\overline{x}^{p\tilde{q}}(y).P$, and $x^{pq}(y).(P \parallel Q)$, where it is bound only in P (i.e., Q does not receive the session y, but can keep communicating on x). Free names of a process P (noted $\mathrm{fn}(P)$) are the set of free names of sessions appearing in P.

As for other process calculi, several syntactically different terms may denote the very same process. To simplify these alternative presentations, we introduce the usual notions of contexts and syntactic congruence.

Definition 3.1 (Contexts) The contexts are defined as follows:

$$\mathcal{C}[\_] ::= \_ \big| (\nu x)\mathcal{C}[\_] \big| (\mathcal{C}[\_] \big|_x P) \big| (P \big|_x \mathcal{C}[\_])$$

Definition 3.2 (Equivalence $\equiv$) The relation $\equiv$ is the smallest equivalence relation closed under contexts (that is, $P \equiv Q \Rightarrow \mathcal{C}[P] \equiv \mathcal{C}[Q]$) satisfying the following rules (where x, y and z are different session names):

$$\begin{aligned}
P\big|_x Q &\equiv Q\big|_x P\\
(P\big|_x Q)\big|_x R &\equiv P\big|_x (Q\big|_x R)\\
P\big|_x \mathrm{close}(x) &\equiv P\\
((\nu x)P)\big|_z Q &\equiv (\nu x)(P\big|_z Q) \quad x \notin \mathrm{fn}(Q)\\
(\nu x)(\nu y)P &\equiv (\nu y)(\nu x)P\\
(P\big|_x Q)\big|_y R &\equiv P\big|_x (Q\big|_y R) \quad x \notin \mathrm{fn}(R),\ y \notin \mathrm{fn}(P)
\end{aligned}$$

Definition 3.3 (Reduction for processes) The actions α for processes are:

$$\alpha ::= x : p \to \tilde{q} : A \big| x : p \to \tilde{q} : \langle\cdot\rangle \big| x : p \to \tilde{q} : \&l \big| \oplus \big| \tau$$

We may write $x : \gamma$ for either $x : p \to \tilde{q} : \langle\cdot\rangle$, $x : p \to \tilde{q} : \&l$, or $x : p \to \tilde{q} : A$.

We note $P \xrightarrow{\alpha} Q$ for a transition from P to Q under the action α. This relation is defined by the rules in Fig. 2.

Most rules are standard and reflect the intuitive explanation of constructs given above. In rule (comm), communication follows a "call-by-name" approach, i.e., M is not evaluated before sending.[2] In reduction (send), we observe that the fresh session handle y is restricted to R and the P_i's, so if y appears in any Q_i, the result would be ill formed. However, as we will see, the typing rules will ensure y does not appear in any Q_i. Similarly, in reduction (wait), corresponding to the definitive closing of a (private) session, can happen only if $x \notin \mathrm{fn}(P)$, but again, the typing rules will ensure x does not escape its scope when reducing $(\nu x)(\mathrm{wait}(x).P)$ into P.

Example 3.1 As a running example, let us consider three participants p, q, r. p chooses whether to send a message to r or not; this choice is communicated to r through an intermediate participant q.

$$\begin{aligned}
P_p &:= \text{if } M \text{ then } \overline{x}^{pq} \triangleright \mathbf{t}.\overline{x}^{pr}(y).\mathrm{wait}(y).\mathrm{close}(x) \text{ else } \overline{x}^{pq} \triangleright \mathbf{f}.\mathrm{close}(x)\\
P_q &:= x^{qp} \triangleleft \{\mathbf{t} : \overline{x}^{qr} \triangleright \mathbf{ok}.\mathrm{close}(x), \mathbf{f} : \overline{x}^{qr} \triangleright \mathbf{quit}.\mathrm{close}(x)\}\\
P_r &:= x^{rq} \triangleleft \{\mathbf{ok} : x^{rp}(y)(\mathrm{close}(y) \parallel \mathrm{close}(x)), \mathbf{quit} : \mathrm{close}(x)\}
\end{aligned}$$

Here is an example of execution:

$$\begin{aligned}
P_p\big|_x P_q\big|_x P_r &\xrightarrow{\oplus} && \overline{x}^{pq} \triangleright \mathbf{t}.\overline{x}^{pr}(y).\mathrm{wait}(y).\mathrm{close}(x)\big|_x P_q\big|_x P_r\\
&\xrightarrow{x:p\to q:\&\mathbf{t}} && \overline{x}^{pr}(y).\mathrm{wait}(y).\mathrm{close}(x)\big|_x \overline{x}^{qr} \triangleright \mathbf{ok}.\mathrm{close}(x)\big|_x P_r\\
&\xrightarrow{x:q\to r:\&\mathbf{ok}} && \overline{x}^{pr}(y).\mathrm{wait}(y).\mathrm{close}(x)\big|_x x^{rp}(y)(\mathrm{close}(y) \parallel \mathrm{close}(x))\\
&\xrightarrow{x:p\to r:\langle\cdot\rangle} && (\nu y)(\mathrm{close}(y)\big|_y \mathrm{wait}(y).\mathrm{close}(x))\big|_x \mathrm{close}(x)\\
&\equiv && (\nu y)(\mathrm{wait}(y).\mathrm{close}(x))\\
&\xrightarrow{\tau} && \mathrm{close}(x)
\end{aligned}$$

We can see that processes with the operation $|_x$ have the structure of a commutative monoid (whose unit is $\mathrm{close}(x)$), thus we will use $\Pi_i^x P_i$ as a shorthand for $P_1\big|_x \dots \big|_x P_n$, omitting n for the sake of readability.

Operational Semantics The semantics of the process calculus is given by a labelled reduction system $P \xrightarrow{\alpha} Q$, where the label α allows to observe the kind of interaction: synchronization (with communication), external choice, internal choice, or internal (unobservable) move.

There is another possible execution, which is:

$$\begin{aligned}
P_p\big|_x P_q\big|_x P_r &\xrightarrow{\oplus} && \overline{x}^{pq} \triangleright \mathbf{f}.\mathrm{close}(x)\big|_x P_q\big|_x P_r\\
&\xrightarrow{x:p\to q:\&\mathbf{f}} && \overline{x}^{qr} \triangleright \mathbf{quit}.\mathrm{close}(x)\big|_x P_r\\
&\xrightarrow{x:q\to r:\&\mathbf{quit}} && \mathrm{close}(x)
\end{aligned}$$

[2] It is possible to consider also a "call-by-value" semantics, where M is evaluated before the communication; but then, for proving the progress property, evaluation must be normalizing for all types, and not only for Booleans.

Fig. 2 Reduction for processes (where $\tilde{q} = \{q_1, \ldots, q_n\}$ and i ranges over $1..n$)

$$\overline{x}^{p\tilde{q}}\langle M\rangle.R \mid_x \Pi_i^x(x^{q_i p}(y_i : A).Q_i) \xrightarrow{x:p\to\tilde{q}:A} R \mid_x \Pi_i^x Q_i[M/y_i] \qquad \text{(comm)}$$

$$\overline{x}^{p\tilde{q}}(y).R \mid_x \Pi_i^x(x^{q_i p}(y).(P_i \parallel Q_i)) \xrightarrow{x:p\to\tilde{q}:\langle\cdot\rangle} (\nu y)(R \mid_y \Pi_i^y P_i) \mid_x \Pi_i^x Q_i \quad \text{if } \forall i, y \notin \text{fn}(Q_i) \qquad \text{(send)}$$

$$\overline{x}^{p\tilde{q}} \triangleright l_j.R \mid_x \Pi_i^x x^{q_i p} \triangleleft \{\ldots, l_j : Q_{i,j}, \ldots\} \xrightarrow{x:p\to\tilde{q}:\&l_j} R \mid_x \Pi_i^x Q_{i,j} \qquad \text{(case)}$$

$$(\nu x)(\text{wait}(x).P) \xrightarrow{\tau} P \quad \text{if } x \notin \text{fn}(P) \qquad \text{(wait)}$$

$$\frac{M \downarrow tt}{\text{if } M \text{ then } P \text{ else } Q \xrightarrow{\oplus} P}\ (choice_1) \qquad \frac{M \downarrow f\!f}{\text{if } M \text{ then } P \text{ else } Q \xrightarrow{\oplus} Q}\ (choice_2)$$

$$\frac{P \xrightarrow{x:\gamma} Q}{(\nu x)P \xrightarrow{\tau} (\nu x)Q} \qquad \frac{P \xrightarrow{\alpha} Q \quad x \notin \text{fn}(\alpha)}{(\nu x)P \xrightarrow{\alpha} (\nu x)Q}$$

$$\frac{P \xrightarrow{\alpha} Q}{P \mid_x R \xrightarrow{\alpha} Q \mid_x R} \qquad \frac{P \equiv P' \quad P' \xrightarrow{\alpha} Q' \quad Q' \equiv Q}{P \xrightarrow{\alpha} Q}$$

Notice that p can start the session with just q and then wait for input from r:

$$P_p\big|_x P_q \xrightarrow{\oplus} \overline{x}^{pq} \triangleright \mathbf{f}.\text{close}(x)\big|_x P_q \xrightarrow{x:p\to q:\&\mathbf{f}} \overline{x}^{qr} \triangleright \mathbf{quit}.\text{close}(x)$$

Example 3.2 A second running example is the well-known two-buyers problem, as in [17]. First Buyer 1 sends a book title to Seller, then Seller sends back a quote to both Buyer 1 and Buyer 2, then Buyer 1 tells Buyer 2 how much she wants to contribute, and Buyer 2 tells Seller if she accepts the quote or not. If the deal is drawn, Seller tells Buyer 2 the expected delivery date at her address.

To formalize this example, we need four base types: bool, int, string, and date. Buyer 2 knows a value $address$: string, and seller knows a value $price$: string $\to$ int and $delivery$: string $\to$ date (the delivery date). There are also the global functions $\leq$: int $\to$ int $\to$ bool and $/, -$: int $\to$ int $\to$ int. We assume all of these values (local and global) are in an environment E. Then, the following are the processes P_{b_1}, P_{b_2}, P_s of Buyer 1, Buyer 2 and Seller, respectively:

4 Partial multiparty session types

In this section, we introduce *partial multiparty session types* (or just "session types"), which will be used to define the behavior of a partial session. The syntax of *messages m* and *session types G* is as follows:

$$m ::= A \big| \,\&\, l \big| \langle G\rangle$$
$$G ::= p \to \tilde{q} : m; G \big| G \oplus G \big| G \,\&\, G \big| \text{end} \big| \text{close} \big| 0 \big| \omega$$

We pose $\mathfrak{G}$ the set of session types with no occurrence of 0 and ω, and $\mathfrak{G}_\omega$ the set of all session types. When not differently stated, we will use types from $\mathfrak{G}_\omega$, while $\mathfrak{G}$ will be used in Sect. 5 to type processes. The set of participant names appearing in G is denoted by fn(G).

Informally, $p \to \tilde{q} : m; G$ means that the participant p sends the message m to the participants in $\tilde{q}$, then the session continues with G. This message can be either a term (from the language $\mathfrak{D}$) of type A, or a label l (noted $\&l$); or a fresh handler for a session of type G (noted $\langle G\rangle$). The type end means that the session ends and the process survives, while close means that both the session and the process end.

$$\begin{aligned}
P_{b_1} ::= {} & \overline{x}^{b_1 s}\langle\text{“War and Peace”}\rangle.x^{b_1 s}(quote : \text{int}).\overline{x}^{b_1 b_2}\langle quote/2\rangle.\text{close}(x)\\
P_{b_2} ::= {} & x^{b_2 s}(quote : \text{int}).x^{b_2 b_1}(quote' : \text{int}).\\
& \quad \text{if } (quote - quote' \leq 100)\\
& \quad \text{then } \overline{x}^{b_2 s} \triangleright \mathbf{ok}.\overline{x}^{b_2 s}\langle address\rangle.x^{b_2 s}(d : \text{date}).\text{close}(x)\\
& \quad \text{else } \overline{x}^{b_2 s} \triangleright \mathbf{quit}.\text{close}(x)\\
P_s ::= {} & x^{s b_1}(title : \text{string}).\overline{x}^{s b_1 b_2}\langle price\ title\rangle.\\
& \quad x^{s b_2} \triangleleft \{\mathbf{ok} : x^{s b_2}(a : string).\overline{x}^{s b_2}\langle delivery\ a\rangle.\text{close}(x),\\
& \qquad \mathbf{quit} : \text{close}(x)\}
\end{aligned}$$

$G_1 \oplus G_2$ (resp. $G_1 \,\&\, G_2$) denotes an *internal* (resp. *external*) choice. Internal choices are made by local participants of the session, contrary to external choices; notice that, in contrast with standard practice, sending or receiving a label $\&l$ is unrelated from the choices done with $\oplus$ or $\&$. Finally, we add the *empty* type 0, which denotes no possible executions (and it is the unit of $\oplus$), and the *inconsistent* type ω, which denotes an error in the session.

Example 4.1 Continuing our running Example 3.1, the following are the types of session x for each process P_p, P_q, P_r by the type system we will present in Sect. 5.

$$G_p := (p \to q : \&\mathbf{t};\ p \to r : \langle \text{end} \rangle;\ \text{close}) \oplus (p \to q : \&\mathbf{f};\ \text{close})$$
$$G_q := (p \to q : \&\mathbf{t};\ q \to r : \&\mathbf{ok};\ \text{close}) \,\&\, (p \to q : \&\mathbf{f};\ q \to r : \&\mathbf{quit};\ \text{close})$$
$$G_r := (q \to r : \&\mathbf{ok};\ p \to r : \langle \text{close} \rangle;\ \text{close}) \,\&\, (q \to r : \&\mathbf{quit};\ \text{close})$$

We will also be able to type compositions of these processes, e.g., the types of x in $P_p \big|_x P_q$ and $P_q \big|_x P_r$ are the following:

$$\begin{aligned} G_{p,q} :=&(p \to q : \&\mathbf{t};\ q \to r : \&\mathbf{ok};\ p \to r : \langle \text{end} \rangle;\ \text{close}) \\ &\oplus (p \to q : \&\mathbf{f};\ q \to r : \&\mathbf{quit};\ \text{close}) \\ G_{q,r} :=&(p \to q : \&\mathbf{t};\ q \to r : \&\mathbf{ok};\ p \to r : \langle \text{close} \rangle;\ \text{close}) \\ &\,\&\, (p \to q : \&\mathbf{f};\ q \to r : \&\mathbf{close};\ \text{close}) \end{aligned}$$

Notice that $G_{p,q}$ describes also the behavior of $P_p \big|_x P_q \big|_x P_r$. As we will see in the next section, these types can be compositionally derived from G_p, G_q, G_r.

Example 4.2 (Two-buyers protocol) Continuing Example 3.2, we define

$$G_1 := b_2 \to s : \mathbf{ok};\ b_2 \to s : \text{string};\ s \to b_2 : \text{date};\ \text{close}$$
$$G_2 := b_2 \to s : \mathbf{quit};\ \text{close}$$

We expect that in the environment E, for the three processes P_{b_1}, P_{b_2}, P_s, session x will have the following types, respectively:

$$G_{b_1} := b_1 \to s : \text{string};\ s \to b_1 : \text{int};\ b_1 \to b_2 : \text{int};\ \text{close}$$
$$G_{b_2} := s \to b_2 : \text{int};\ b_1 \to b_2 : \text{int};\ (G_1 \oplus G_2)$$
$$G_s := b_1 \to s : \text{string};\ s \to b_1, b_2 : \text{int};\ (G_1 \,\&\, G_2)$$

In the end, the whole process $P_s \big|_x P_{b_1} \big|_x P_{b_2}$ will communicate on a session x following the type $b_1 \to s : \text{string};\ s \to b_1, b_2 : \text{int};\ b_1 \to b_2 : \text{int};\ (G_1 \oplus G_2)$.

Now, we aim to define when two session types are equivalent from the point of view of a set of participants. The idea is that a subset of all participants involved in a session may collect a limited knowledge of the events happening in the session, thus several possible types can be compatible with this limited knowledge—which can be already enough to tell possible incoherences. The events we can observe are *communications* C and are defined as follows:

$$C ::= p \to \tilde{q} : m \big| \text{end} \big| \text{close} \big| 0 \big| \omega \big| 1$$

We denote by $\mathfrak{C}_\omega$ the set of all communications, and by $\mathfrak{C} = \mathfrak{C}_\omega \setminus \{\omega, 0\}$ the set of *executable* communications. The communications end, close, 0, ω are called *terminal*; the only non-terminal communications are $p \to \tilde{q} : m$ and 1, the latter representing any communication which is not observable from the current process. Thus, we can see terminal communications as types, and non-terminal communications as prefixes of types; in particular 1 is a "neutral" prefix for session types, i.e., for all G, we define $1; G = G$.

In the following, we denote by $S, S_1, \ldots \subset \mathfrak{P}$ finite sets of participants, which we call *viewpoints*.

Definition 4.1 (Independence relation) Let S be a viewpoint. The *independence of communications* relative to S is the smallest symmetric relation $I_S \subseteq \mathfrak{C}_\omega \times \mathfrak{C}_\omega$ such that $C\ I_S\ 1$ for any C, and $(p \to \tilde{q} : m)\ I_S\ (p' \to \tilde{q}' : m')$ whenever $((\{p\} \cup \tilde{q}) \cap (\{p'\} \cup \tilde{q}')) \cap S = \varnothing$.

Informally, $C_1\ I_S\ C_2$ means that the common participants of C_1 and C_2 are not in S. This independence is relative to the viewpoint S, because when $C_1\ I_S\ C_2$, the participants in S cannot discriminate between $C_1; C_2; G$ and $C_2; C_1; G$. In fact, we can define an equivalence relation between session types relative to S.

Definition 4.2 (Equivalence relation) For any set of participants S, we define the relation $\simeq_S \subseteq \mathfrak{G}_\omega \times \mathfrak{G}_\omega$ on session types as the smallest congruence satisfying the axioms in Fig. 3.

We can see that the operations $\oplus$ and $\&$, together with the constants 0 and ω, form a unital commutative semiring. We note $\bigoplus\{G_1, \ldots, G_n\}$ for $G_1 \oplus \ldots \oplus G_n$ and $\bigwith\{G_1, \ldots, G_n\}$ for $G_1 \,\&\, \ldots \,\&\, G_n$; in particular, $\bigoplus \varnothing = 0$ and $\bigwith \varnothing = \omega$. The axiom (OOOE) allows an "out of order" execution of independent communications. Notice that in general $G \oplus \omega \not\simeq_S \omega$ because the behavior of a process of type $G \oplus \omega$ is not necessarily always inconsistent.[3]

[3] In fact, the equivalence $G \oplus \omega \simeq_S \omega$ would invalidate the semantic interpretation of types given in Sect. 7, and in particular Theorem 7.1.

Fig. 3 Congruence equivalence for session types

$$G \oplus 0 \simeq_S G \qquad G_1 \oplus (G_2 \oplus G_3) \simeq_S (G_1 \oplus G_2) \oplus G_3$$
$$G \oplus G \simeq_S G \qquad G_1 \oplus G_2 \simeq_S G_2 \oplus G_1$$
$$G \mathbin{\&} \omega \simeq_S G \qquad G_1 \mathbin{\&} (G_2 \mathbin{\&} G_3) \simeq_S (G_1 \mathbin{\&} G_2) \mathbin{\&} G_3$$
$$G \mathbin{\&} G \simeq_S G \qquad G_1 \mathbin{\&} G_2 \simeq_S G_2 \mathbin{\&} G_1$$
$$G \mathbin{\&} 0 \simeq_S 0 \qquad G_1 \mathbin{\&} (G_2 \oplus G_3) \simeq_S (G_1 \mathbin{\&} G_2) \oplus (G_1 \mathbin{\&} G_3)$$
$$C; \omega \simeq_S \omega \qquad C; (G_1 \mathbin{\&} G_2) \simeq_S (C; G_1) \mathbin{\&} (C; G_2)$$
$$C; 0 \simeq_S 0 \qquad C; (G_1 \oplus G_2) \simeq_S (C; G_1) \oplus (C; G_2)$$
$$C_1; C_2; G \simeq_S C_2; C_1; G \quad \text{if } C_1 \; I_S \; C_2 \qquad \text{(OOOE)}$$

The equation $C; 0 \simeq_S 0$ corresponds to the fact that 0 means "no possible executions," not even the stuck one (as we will see in Sect. 7, the interpretation of 0 is the empty set); thus, prepending an execution to nothing yields nothing.

The fact that $G_1 \oplus G_2$ or $G_1 \mathbin{\&} G_2$ are unrelated from the action of sending a choice allows us to move these operators around without changing the meaning of the type. Hence, we can introduce *disjunctive normal forms* of session types.

Definition 4.3 (Disjunctive Normal Form) A *chain of communications* is a session type of the form $C_1; \ldots; C_n$ (where C_n is necessarily terminal).

A session type G is in Disjunctive Normal Form (DNF) if it is of the form $G = \bigoplus\{\bigwith A_1, \ldots, \bigwith A_n\}$ where each A_i is a set of chains of communications where for every message $\langle G' \rangle$, G' is in DNF.

In DNF, a type can be seen as a set of sets of traces (chains of communications), the intuition being that a trace describes a single possible interaction of a process. A set of traces defines a deterministic strategy followed by a single process P, describing how P reacts for any possible choice from other processes. A set of sets of traces describes all the possible strategies that P can follow once it has selected all its possible internal choices. So, describing a behavior in DNF is like saying that a process P starts by anticipating all possible internal choices for all possible interactions during execution. After that, P becomes deterministic and reacts in a single possible way to communications of other processes.

The equivalence relation on types allows us to rewrite any type in a DNF.

Proposition 4.1 *For any type G and set of participants S, we can compute a G' in DNF such that $G' \simeq_S G$.*

Proof Given a type G, we can look at subterms inside G of the forms $G_1 \mathbin{\&} (G_2 \oplus G_3)$, $C; (G_1 \mathbin{\&} G_2)$, $C; (G_1 \oplus G_2)$. Each of them can be replaced by the corresponding equivalent subterm, i.e., $(G_1 \mathbin{\&} G_2) \oplus (G_1 \mathbin{\&} G_3)$, $(C; G_1) \mathbin{\&} (C; G_2)$, $(C; G_1) \oplus (C; G_2)$. By repeatedly applying these rewritings, all $\oplus$ are moved at the top level and the ";" at the leaves of the type. □

5 Type system

In this section, we introduce the type system for processes. A key point of our approach is that types have always to be considered with respect to a set of participants, i.e., a viewpoint.

Definition 5.1 (Environment) A *typing declaration* for session x is a triple $x : \langle G \big| S \rangle$ where $G \in \mathfrak{G}$ and $S \subseteq \mathfrak{P}$. S is the set of *local participants* of x.

A *(session typing) environment* Γ is a finite set of typing declarations

$$\Gamma = x_1 : \langle G_1 \big| S_1 \rangle, \ldots, x_n : \langle G_n \big| S_n \rangle$$

such that $x_1, \ldots x_n$ are all distinct.

The main differences between our environments and those in [11,29] are that session types replace local types, and each session is endowed with a set of local participants, in addition to its session type.

We can extend internal choices and type equivalences to environments.

Definition 5.2 Let $\Gamma = x_1 : \langle G_1 \big| S_1 \rangle, \ldots, x_n : \langle G_n \big| S_n \rangle$ and $\Gamma' = x_1 : \langle G'_1 \big| S_1 \rangle, \ldots, x_n : \langle G'_n \big| S_n \rangle$ be two environments with the same domain and same sets of local participants. In this case, we define

$$\Gamma \oplus \Gamma' := x_1 : \langle G_1 \oplus G'_1 \big| S_1 \rangle, \ldots, x_n : \langle G_n \oplus G'_n \big| S_n \rangle.$$

Definition 5.3 (Equivalent environments) We define $\simeq$ on environments as the smallest equivalence relation satisfying the following rule:

$$\frac{\Gamma_1 \simeq \Gamma_2 \qquad G_1 \simeq_S G_2}{\Gamma_1, x : \langle G_1 \big| S \rangle \simeq \Gamma_2, x : \langle G_2 \big| S \rangle}$$

We can now introduce the typing judgment for processes $E; P \vdash \Gamma$, where E is a term typing environment and Γ is a session typing environment. Intuitively, it means "under the declarations in E, for each $x : \langle G \big| S \rangle$ in Γ, the participants of P that interact on x are S and they follow the behavior G."

Fig. 4 Type system for processes

$$\frac{E \vdash_{\mathcal{D}} M : A \quad E; P \vdash \Gamma, x : \langle G \mid p\rangle}{E; \overline{x}^{p\tilde{q}}\langle M\rangle.P \vdash \Gamma, x : \langle p \to \tilde{q} : A; G \mid p\rangle}\ (sendvalue)$$

$$\frac{E, y : A; P \vdash \Gamma, x : \langle G \mid q\rangle \quad q \in \tilde{q}}{E; x^{qp}(y : A).P \vdash \Gamma, x : \langle p \to \tilde{q} : A; G \mid q\rangle}\ (recvvalue)$$

$$\frac{E; P \vdash \Gamma, y : \langle G_1 \mid p\rangle, x : \langle G_2 \mid p\rangle}{E; \overline{x}^{p\tilde{q}}(y).P \vdash \Gamma, x : \langle p \to \tilde{q} : \langle G_1\rangle; G_2 \mid p\rangle}\ (send)$$

$$\frac{E; P \vdash \Gamma_1, y : \langle G_1 \mid q\rangle \quad E; Q \vdash \Gamma_2, x : \langle G_2 \mid q\rangle \quad q \in \tilde{q}}{E; x^{qp}(y).(P \parallel Q) \vdash \Gamma_1, \Gamma_2, x : \langle p \to \tilde{q} : \langle G_1\rangle; G_2 \mid q\rangle}\ (recv)$$

$$\frac{E; P \vdash \Gamma, x : \langle G \mid p\rangle}{E; \overline{x}^{p\tilde{q}} \triangleright l.P \vdash \Gamma, x : \langle p \to \tilde{q} : \& l; G \mid p\rangle}\ (sel)$$

$$\frac{E; P_1 \vdash \Gamma, x : \langle G_1 \mid q\rangle \quad \cdots \quad E; P_n \vdash \Gamma, x : \langle G_n \mid q\rangle \quad q \in \tilde{q}}{E; x^{qp} \triangleleft \{l_1 : P_1, \ldots, l_n : P_n\} \vdash \Gamma, x : \langle (p \to \tilde{q} : \& l_1; G_1) \,\&\, \ldots \,\&\, (p \to \tilde{q} : \& l_n; G_n) \mid q\rangle}\ (case)$$

$$\frac{E \vdash_{\mathcal{D}} M : \text{Bool} \quad E; P \vdash \Gamma_1 \quad E; Q \vdash \Gamma_2}{E; \text{if } M \text{ then } P \text{ else } Q \vdash \Gamma_1 \oplus \Gamma_2}\ (\oplus)$$

$$\frac{}{E; \text{close}(x) \vdash x : \langle \text{close} \mid \varnothing\rangle}\ (close) \qquad \frac{E; P \vdash \Gamma}{E; \text{wait}(x).P \vdash \Gamma, x : \langle \text{end} \mid \varnothing\rangle}\ (wait)$$

$$\frac{E; P \vdash \Gamma, x : \langle G \mid S\rangle \quad G \downarrow S}{E; (\nu x)P \vdash \Gamma}\ (\nu) \qquad \frac{E; P \vdash \Gamma \quad \Gamma \simeq \Gamma'}{E; P \vdash \Gamma'}\ (\simeq)$$

$$\frac{E; P \vdash \Gamma, x : \langle G \mid S_1\rangle \quad S_2 \cap \text{fn}(G) = \varnothing}{E; P \vdash \Gamma, x : \langle G \mid S_1 \cup S_2\rangle}\ (extra)$$

$$\frac{E; P \vdash \Gamma_1, x : \langle G_1 \mid S_1\rangle \quad E; Q \vdash \Gamma_2, x : \langle G_2 \mid S_2\rangle \quad S_1 \cap S_2 = \varnothing \quad G \simeq_{S_1 \uplus S_2} G_1 \ {}^{S_1}\!\vee^{S_2}\, G_2}{E; P \mid_x Q \vdash \Gamma_1, \Gamma_2, x : \langle G \mid S_1 \uplus S_2\rangle}\ (|)$$

The typing rules are shown in Fig. 4. Rules (*sendvalue*), (*recvvalue*), (*send*), (*recv*), (*sel*$_i$), and (*case*) deal with communication. Differently from most type systems (see, e.g., [11]), the send and receive actions are typed by the same global type, and not by dual types: in our approach the duality is given by the set of participants, i.e., the viewpoint, which is either the sender or the receiver.

Rule ($\oplus$) types an internal choice between two processes. This choice is propagated to all sessions where the process is involved, so the sum is done componentwise between the types of the two branches, as per Definition 5.2. If the internal choice is irrelevant for some session x (i.e., we have $x : \langle G \big| S\rangle$ in both premises) then in the conclusion we would have $x : \langle G \oplus G \big| S\rangle$, which is equivalent to the former. We can rewrite types into equivalent ones with rule ($\simeq$).

Rules (*close*) and (*wait*) correspond, respectively, to the 1 and $\perp$ rules in linear logic, and they both assume there is no named participant, therefore the set of participants in the conclusion is empty.

Rule (ν) allows us to create a local, restricted session. To correctly type the local session, we need to check that its type is complete with respect to the current participants, since no other participants will be able to join that session afterward.

$$\frac{\{p\} \cup \tilde{q} \subseteq S \quad G \downarrow S}{p \to \tilde{q} : A; G \downarrow S} \qquad \frac{\{p\} \cup \tilde{q} \subseteq S \quad G_1 \downarrow \{p\} \cup \tilde{q} \quad G_2 \downarrow S}{p \to \tilde{q} : \langle G_1\rangle; G_2 \downarrow S}$$

$$\frac{\{p\} \cup \tilde{q} \subseteq S \quad G \downarrow S}{p \to \tilde{q} : \& l; G \downarrow S} \qquad \frac{G_1 \downarrow S \quad G_2 \downarrow S}{G_1 \oplus G_2 \downarrow S} \qquad \frac{G_1 \downarrow S \quad G_2 \downarrow S}{G_1 \,\&\, G_2 \downarrow S}$$

$$\frac{}{\text{end} \downarrow S} \qquad \frac{}{0 \downarrow S} \qquad \frac{G_1 \downarrow S \quad G_1 \simeq_S G_2}{G_2 \downarrow S}$$

Fig. 5 Rules for the finalized judgment

To this end, we introduce the notion of *finalized* session type. Intuitively, a type G is finalized for a viewpoint S, denoted $G \downarrow S$, if all participants involved in the type are in the viewpoint, there are no occurrence of ω or close (because we need to avoid deadlocks and miscommunications), and that the end of the session is not the end of the process (because we are within a subsession). The rules for the auxiliary judgment $G \downarrow S$ are in Fig. 5.

The (*extra*) rule allows us to add participants which actually do not interact with the sessions; this is needed for the subject reduction.

Finally, rule ($|$) is one of the key novelties of our type system. This rule allows us to connect two processes through a shared session x *merging* their respective types and viewpoints (which must be separated). The type G of the shared session is computed compositionally from G_1, S_1 and G_2, S_2, by $G_1 \,{}^{S_1}\!\vee^{S_2}\, G_2$. The definition of this operator is quite complex and is postponed to Sect. 6; for the moment it is enough to know that it is associative and symmetric, and the resulting type G is a "minimal" type from the viewpoint $S_1 \uplus S_2$ such that it is equivalent to G_1 from the viewpoint S_1 and to G_2 from the viewpoint S_2. Moreover, if G_1, G_2 are not compatible (e.g., due to a deadlock or a miscommunication) the type $G_1 \,{}^{S_1}\!\vee^{S_2}\, G_2$ will contain some occurrence of ω, and thus $G_1 \,{}^{S_1}\!\vee^{S_2}\, G_2 \notin \mathfrak{G}$. To guarantee that only valid types are used for the merged session, rule $(|)$ requires to find $G \in \mathfrak{G}$ such that $G \simeq_{S_1 \uplus S_2} G_1 \,{}^{S_1}\!\vee^{S_2}\, G_2$; hence, the parallel composition of incompatible processes cannot be typed.

Remark 5.1 It may be interesting to compare our rule for parallel composition with the cut rule for linear logic [15], that for binary session types [32], and that for multiparty session types [11]:

$$\frac{\vdash \Gamma, G \quad \vdash \Delta, G^{\perp}}{\vdash \Gamma, \Delta}(Cut)$$

$$\frac{P \vdash \Gamma, x : G \quad Q \vdash \Delta, x : G^{\perp}}{(\nu x : G)(P \,\big|\, Q) \vdash \Gamma, \Delta}(BinPar)$$

$$\frac{P_i \vdash \Gamma_i, x^{p_i} : A_i \quad G \vDash \{p_i : A_i\}_i}{(\nu x : G)(\Pi_i^x P_i) \vdash \{\Gamma_i\}_i}(MultiPar)$$

Each of these rules corresponds to the applications of two rules of our system: the rule $(|)$ which merges partial sessions, and the rule (ν) which closes the session. This correspondence, albeit in a logical setting and for binary choreographies only, has been previously observed in [10], where the (Cut) rule is split into two rules (called (Conn) and (Scope)). For instance, if we assume that A_1, A_2, and G are suitable session types, we have the following derivation of the rule for binary session types $(BinPar)$ (we omit the environment E for the sake of simplicity):

$$\frac{\dfrac{P \vdash \Gamma, x{:}\langle A_1 \,\big|\, S_1\rangle \quad Q \vdash \Delta, x{:}\langle A_2 \,\big|\, S_2\rangle \quad G \simeq_{S_1 \uplus S_2} A_1 \,{}^{S_1}\!\vee^{S_2} A_2}{P \,\big|_x\, Q \vdash \Gamma, \Delta, x{:}\langle G \,\big|\, S_1 \uplus S_2\rangle} \qquad G{\downarrow} S_1 \uplus S_2}{(\nu x)(P \,\big|_x\, Q) \vdash \Gamma, \Delta}$$

In the case of a multiparty session involving n participants, we can apply $(|)$ $n-1$ times, and then the (ν) rule to close the session. Notice that the compatibility check in the premises of $(MultiPar)$ corresponds to a sequence of $n-1$ binary merges followed by the finalization check. This allows to spot incompatible processes (e.g., due to a deadlock) as soon as possible: $A_1 \,{}^{S_1}\!\vee^{S_2}\, A_2$ would yield a type containing some ω, and hence for no $G \in \mathfrak{G}$ it would be $G \simeq_{S_1 \uplus S_2} A_1 \,{}^{S_1}\!\vee^{S_2}\, A_2$.

6 Merging partial session types

The central part of the type system is the merging algorithm that infers the result of interaction of two partial session types. In this section, we will define the merge function $G_1 \,{}^{S_1}\!\vee^{S_2}\, G_2$, where G_1 and G_2 describe the behavior of a session from the viewpoint of the local participants in the sets S_1 and S_2, respectively. $G_1 \,{}^{S_1}\!\vee^{S_2}\, G_2$ then describes the behavior of the session from the unified viewpoint $S_1 \cup S_2$. In particular, if G_1 and G_2 are incompatible then $G_1 \,{}^{S_1}\!\vee^{S_2}\, G_2$ contains some occurrence of ω. [4]

To merge two types, we can consider them in DNF; in this way we can recursively reduce the problem to merging chains of communications, by means of a function $\mathsf{mcomm}_{S_1,S_2}(C_1, C_2)$. Informally, we merge two sequences of communications by considering all possible reorderings which are compatible with each other. This give us a set of all possible merged behaviors, which we glue together using external choices (&). Finally, for merging general types in DNF, we proceed by recursion until we have to merge sequences of communications. Thus, two types are compatible if they can agree on at least a pair of merged sequences of communications, whatever their internal choices; if no such sequences exist, we get ω as a result. Extra complexity is given by the fact that when we have to merge two communications of the form $p \to \tilde{q} : \langle G_1\rangle$, $p \to \tilde{q} : \langle G_2\rangle$, we have to merge also G_1 and G_2; therefore, the function $\mathsf{mcomm}_{S_1,S_2}(C_1, C_2)$ and the function $G_1 \,{}^{S_1}\!\vee^{S_2}\, G_2$ for merging session types are mutually recursive.

Before diving into the definition of mcomm_{S_1,S_2} and ${}^{S_1}\!\vee^{S_2}$, let us mention that we will also need the following helper functions and predicates (which we will illustrate in Example 6.1 later on):

- the *continuation* partial function $\mathsf{cont}_S(G, C)$, which takes a chain of communications G and a communication C as input, and returns a type that corresponds (up to $\simeq_S$) to what remains in G after having executed C;
- the *mergeability* predicate $C_1 \,{}^{S_1}\!\heartsuit^{S_2}\, C_2$ tells us whether two communications C_1 and C_2 are mergeable, from their respective viewpoints S_1, S_2;
- the *synchronization* total function $\mathsf{sync}_{S_1,S_2}(f)(G_1, G_2)$ takes a (partial) function $f : \mathfrak{C} \times \mathfrak{C} \rightharpoonup \mathfrak{C}$, called *merging function* and describing how two communications can be merged, and two chains of communications G_1 and G_2; it returns the set of all possible tuples (C_1, G_1', C_2, G') such that $C_1; G_1' \simeq_{S_1} G_1$, $C_2; G_2' \simeq_{S_2} G_2$ and C_1 and C_2 are mergeable (according to f);
- finally, the partial function $\mathsf{map}_{S_1,S_2}(f)(G_1, G_2)$ takes a (partial) function $f : \mathfrak{C} \times \mathfrak{C} \rightharpoonup \mathfrak{C}$ and two session types

[4] A prototype implementation of the merging algorithm can be found at https://github.com/cstolze/partial-session-types-prototype.

in DNF as arguments, and maps f on the pair (G_1, G_2); the result is a session type obtained by merging G_1 and G_2 according to f (i.e., where possible).

With these functions and predicates, we will be able to define $\mathsf{mcomm}_{S_1,S_2}(C_1, C_2)$ and $G_1 \ {}^{S_1}\!\vee^{S_2}\ G_2$. These functions are non-deterministic, but $G_1 \ {}^{S_1}\!\vee^{S_2}\ G_2$ is deterministic up to $\simeq$ (see Theorem 7.2).

In order to prove termination of these functions, we need to introduce the *length* l of session types, and the *height* h of communications and session types as the maximal number of nested subsessions. Formally, we have:

$$\begin{aligned}
l(G_1 \mathbin{\&} G_2) &:= l(G_1) + l(G_2) & h(G_1 \mathbin{\&} G_2) &:= \max(h(G_1), h(G_2))\\
l(G_1 \oplus G_2) &:= l(G_1) + l(G_2) & h(G_1 \oplus G_2) &:= \max(h(G_1), h(G_2))\\
l(C; G) &:= 1 + l(G) & h(C; G) &:= \max(h(C), h(G))\\
l(G) &:= 1 \quad \text{otherwise.} & h(p \to \tilde{q} : \langle G\rangle) &:= 1 + h(G)\\
& & h(C) &:= 0 \quad \text{otherwise.}
\end{aligned}$$

6.1 Mapping merging functions over session types

Definition 6.1 (cont) The partial function $\mathsf{cont}_S(G, C)$ takes as input a chain of communications G and a communication C, and returns some G' in DNF such that $G \simeq_S C; G'$. It is undefined if such G' does not exist.

Intuitively, $\mathsf{cont}_S(G, C)$ is a kind of Brzozowski derivative that tells us what happens in G after the communication C.

Proposition 6.1 *The function* cont *is computable, and moreover* $l(C; G') = l(G)$ *and* $h(C; G') = h(G)$.

Note that $\mathsf{dom}(\mathsf{cont}_S(G, \_))$ is finite, and can be computed using the "out-of-order execution" axiom (OOOE, Definition 4.2) repeatedly. This domain set represents the immediate communications that G allows for.

Definition 6.2 (sync) Let G_1, G_2 be chains of communications in DNF.

We then define $\mathsf{sync}_{S_1,S_2}(f)(G_1, G_2)$ as the following set:

$$\begin{aligned}
&\{(C_1, C_2, G_1', G_2') \,|\, G_1' = \mathsf{cont}_S(G_1, C_1),\\
&G_2' = \mathsf{cont}_S(G_2, C_2),\\
&(C_1, C_2) \neq (1, 1),\ f(C_1, C_2) \text{ is defined}\}.
\end{aligned}$$

Intuitively, $\mathsf{sync}_{S_1,S_2}(f)(G_1, G_2)$ returns a set containing all possible pairs of immediate communications C_1, C_2 in G_1, G_2, respectively, that can be merged (according to f), as well as their continuations. Recall that 1 represents any communication which is not observable from the given viewpoint.

Definition 6.3 (Function map) Let S_1, S_2 be two sets of participants, $G_1, G_2 \in \mathfrak{C}$ two types in DNF and $f : \mathfrak{C} \times \mathfrak{C} \rightharpoonup \mathfrak{C}$ a partial map such that for any C_1, C_2:

- $f(C_1, C_2)$ is termina+l if and only if it is defined and C_1, C_2 are both terminal
- if $f(C_1, C_2)$ is defined then either both or none of C_1 and C_2 are terminal.

Then, $\mathsf{map}_{S_1,S_2}(f)(G_1, G_2)$ is defined recursively over G_1, G_2 as follows:

- First cases:

$$\begin{aligned}
\mathsf{map}_{S_1,S_2}(f)(G_1 \oplus G_2, G_3) &:= \mathsf{map}_{S_1,S_2}(f)(G_1, G_3) \oplus \mathsf{map}_{S_1,S_2}(f)(G_2, G_3)\\
\mathsf{map}_{S_1,S_2}(f)(G_1, G_2 \oplus G_3) &:= \mathsf{map}_{S_1,S_2}(f)(G_1, G_2) \oplus \mathsf{map}_{S_1,S_2}(f)(G_1, G_3)\\
\mathsf{map}_{S_1,S_2}(f)(G_1 \mathbin{\&} G_2, G_3) &:= \mathsf{map}_{S_1,S_2}(f)(G_1, G_3) \mathbin{\&} \mathsf{map}_{S_1,S_2}(f)(G_2, G_3)\\
\mathsf{map}_{S_1,S_2}(f)(G_1, G_2 \mathbin{\&} G_3) &:= \mathsf{map}_{S_1,S_2}(f)(G_1, G_2) \mathbin{\&} \mathsf{map}_{S_1,S_2}(f)(G_1, G_3)
\end{aligned}$$

- If G_1, G_2 are both chains of communications and at least one of them is not a terminal communication, we pose $B := \mathsf{sync}_{S_1,S_2}(f)(G_1, G_2)$ and we have:
  - If G_1 or G_2 ends with 0, $\mathsf{map}_{S_1,S_2}(f)(G_1, G_2) := 0$.
  - If G_1 or G_2 ends with ω, or if $B = \varnothing$, then $\mathsf{map}_{S_1,S_2}(f)(G_1, G_2) := \omega$.
  - Otherwise:

$$\mathsf{map}_{S_1,S_2}(f)(G_1, G_2) := \mathop{\&}\{f(C_1, C_2); \mathsf{map}_{S_1,S_2}(f)(G_1', G_2') \,|\, (C_1, C_2, G_1', G_2') \in B\}$$

- If G_1 and G_2 are both terminal communications, then:

$$\mathsf{map}_{S_1,S_2}(f)(G_1, G_2) := \begin{cases} 0 & \text{if } G_1 \text{ or } G_2 \text{ is } 0\\ f(G_1, G_2) & \text{if } f(G_1, G_2) \text{ is defined}\\ \omega & \text{otherwise.}\end{cases}$$

The two conditions on f guarantee that $\mathsf{map}_{S_1,S_2}(f)(G_1, G_2)$ is well-defined in the last two cases, when f is applied to G_1, G_2 or to the chains C_1, C_2.

Proposition 6.2 *Termination of* map *is ensured by induction on* $l(G_1) + l(G_2)$.

Note that, when we computing $\mathsf{map}_{S_1,S_2}(f)(G_1, G_2)$, every application of f is of the form $f_{S_1,S_2}(C_1, C_2)$, where $h(C_1) + h(C_2) \leq h(G_1) + h(G_2)$.

6.2 Merging communications and session types

We now define the partial function $\mathsf{mcomm}_{S_1,S_2}(C_1, C_2)$ which merges compatible communications C_1 (from the viewpoint S_1) and C_2 (from the viewpoint S_2) and returns, if possible, the new communication from the merged viewpoints $S_1 \cup S_2$. We also define by mutual recursion the merging function for session types, which is just a shorthand for map applied to mcomm:

$$G_1 \ {}^{S_1}\!\vee^{S_2}\ G_2 := \mathsf{map}_{S_1,S_2}(\mathsf{mcomm}_{S_1,S_2})(G_1, G_2).$$

We suppose that G_1 and G_2 are in DNFs, but it can be applied to any session types by rewriting them in DNF thanks to Theorem 7.2.

The definition of $\mathsf{mcomm}_{S_1,S_2}(C_1, C_2)$ requires to check whether C_1 (from the viewpoint S_1) and a communication C_2 (from the viewpoint S_2) are actually mergeable. Formally, this notion is defined by the following relation.

Definition 6.4 (Mergeability) We define $C_1 \ {}^{S_1}\!\heartsuit^{S_2}\ C_2$ as follows:

$$\frac{\{p\} \cup \tilde{q}_1 \cup \tilde{q}_2 \subseteq S_1 \cup S_2 \Rightarrow (G_1 \ {}^{S_1}\!\vee^{S_2}\ G_2) \downarrow S_1 \cup S_2 \qquad p \in S_1 \Rightarrow \tilde{q}_2 \subseteq \tilde{q}_1 \quad p \in S_2 \Rightarrow \tilde{q}_1 \subseteq \tilde{q}_2 \quad S_1 \cap \tilde{q}_2 \subseteq \tilde{q}_1 \quad S_2 \cap \tilde{q}_1 \subseteq \tilde{q}_2}{p \to \tilde{q}_1 : \langle G_1 \rangle \ {}^{S_1}\!\heartsuit^{S_2}\ p \to \tilde{q}_2 : \langle G_2 \rangle}$$

$$\frac{p \in S_1 \Rightarrow \tilde{q}_2 \subseteq \tilde{q}_1 \quad p \in S_2 \Rightarrow \tilde{q}_1 \subseteq \tilde{q}_2 \quad S_1 \cap \tilde{q}_2 \subseteq \tilde{q}_1 \quad S_2 \cap \tilde{q}_1 \subseteq \tilde{q}_2}{p \to \tilde{q}_1 : \&l \ {}^{S_1}\!\heartsuit^{S_2}\ p \to \tilde{q}_2 : \&l}$$

$$\frac{p \in S_1 \Rightarrow \tilde{q}_2 \subseteq \tilde{q}_1 \quad p \in S_2 \Rightarrow \tilde{q}_1 \subseteq \tilde{q}_2 \quad S_1 \cap \tilde{q}_2 \subseteq \tilde{q}_1 \quad S_2 \cap \tilde{q}_1 \subseteq \tilde{q}_2}{p \to \tilde{q}_1 : A \ {}^{S_1}\!\heartsuit^{S_2}\ p \to \tilde{q}_2 : A}$$

$$\frac{C_2 \ {}^{S_2}\!\heartsuit^{S_1}\ C_1}{C_1 \ {}^{S_1}\!\heartsuit^{S_2}\ C_2} \qquad \frac{(\{p\} \cup \tilde{q}) \cap S_1 = \varnothing}{1 \ {}^{S_1}\!\heartsuit^{S_2}\ p \to \tilde{q} : m} \qquad \frac{}{1 \ {}^{S_1}\!\heartsuit^{S_2}\ 1}$$

$$\frac{}{\mathrm{close} \ {}^{S_1}\!\heartsuit^{S_2}\ \mathrm{close}} \qquad \frac{}{\mathrm{close} \ {}^{S_1}\!\heartsuit^{S_2}\ \mathrm{end}}$$

The first rule deserves some explanations. In the first hypothesis, G_1 and G_2 describe sessions whose participants can be only in $\{p\}\cup\tilde{q}_1\cup\tilde{q}_2$; if all these participants are in $S_1\cup S_2$, then after the merge all the participants are present and therefore the communication must be safe, because no other participant may join later. This means that, in this case, we have to check that the merge of G_1 and G_2 is finalized. The second hypothesis (and dually the third one) corresponds to the fact that in the $(send)$ rule of Fig. 4, the sender specifies all receiving participants, while in $(recv)$ a receiver may not know about other receivers; therefore, if $p \to \tilde{q}_1 : \langle G_1 \rangle$ describes the communication from the point of view of the sender (i.e., $p \in S_1$), then $\tilde{q}_2$ is a set of receivers only, and must be contained in $\tilde{q}_1$. The fourth (and dually the fifth) hypothesis means that if a participant which is known to a process (i.e., in S_1) appears as receiver for other process (i.e., in $\tilde{q}_2$), then it must appear as a receiver also by the first process.

Notice that ${}^{S_1}\!\heartsuit^{S_2}$ uses $G_1 \ {}^{S_1}\!\vee^{S_2}\ G_2$, so in the end ${}^{S_1}\!\heartsuit^{S_2}$ is defined by mutual recursion together with mcomm.

Definition 6.5 (Function mcomm) If $C_1 \ {}^{S_1}\!\heartsuit^{S_2}\ C_2$, then:

$$\begin{aligned}
\mathsf{mcomm}_{S_1,S_2}(p \to \tilde{q} : \&l, p \to \tilde{q}' : \&l) &:= p \to (\tilde{q} \cup \tilde{q}') : \&l \\
\mathsf{mcomm}_{S_1,S_2}(p \to \tilde{q} : A, p \to \tilde{q}' : A) &:= p \to (\tilde{q} \cup \tilde{q}') : A \\
\mathsf{mcomm}_{S_1,S_2}(p \to \tilde{q} : \langle G_1 \rangle, p \to \tilde{q}' : \langle G_2 \rangle) &:= p \to (\tilde{q} \cup \tilde{q}') : \\
&\quad \langle G_1 \ {}^{S_1}\!\vee^{S_2}\ G_2 \rangle \\
\mathsf{mcomm}_{S_1,S_2}(1, C) &:= C \\
\mathsf{mcomm}_{S_1,S_2}(C, 1) &:= C \\
\mathsf{mcomm}_{S_1,S_2}(C, \mathrm{close}) &:= C \\
\mathsf{mcomm}_{S_1,S_2}(\mathrm{close}, C) &:= C
\end{aligned}$$

Otherwise, $\mathsf{mcomm}_{S_1,S_2}(C_1, C_2)$ is undefined.

Proposition 6.3 *For all C_1, C_2, S_1, S_2, we have that $C_1 \ {}^{S_1}\!\heartsuit^{S_2}\ C_2$ is decidable, and $\mathsf{mcomm}_{S_1,S_2}(C_1, C_2)$ terminates.*

This proposition can be proved by simultaneous induction on $h(C_1) + h(C_2)$.

Example 6.1 Continuing Example 4.1, let us recall the types of participants p, r:

$$\begin{aligned}
G_p &:= G'_p \oplus G''_p & G_r &:= G'_r \,\&\, G'_r \\
G'_p &:= p \to q : \&\mathbf{t};\ p \to r : \langle \mathrm{end} \rangle;\ \mathrm{close} & G''_p &:= p \to q : \&\mathbf{f};\ \mathrm{close} \\
G'_r &:= q \to r : \&\mathbf{ok};\ p \to r : \langle \mathrm{close} \rangle;\ \mathrm{close} & G''_r &:= q \to r : \&\mathbf{quit};\ \mathrm{close}
\end{aligned}$$

We have that:

$$\begin{aligned}
\mathsf{dom}(\mathsf{cont}_{\{p\}}(G'_p, \_)) &= \{p \to q : \&\mathbf{t}\} \\
\mathsf{dom}(\mathsf{cont}_{\{p\}}(G''_p, \_)) &= \{p \to q : \&\mathbf{f}\} \\
\mathsf{dom}(\mathsf{cont}_{\{r\}}(G'_r, \_)) &= \{q \to r : \&\mathbf{ok}\} \\
\mathsf{dom}(\mathsf{cont}_{\{r\}}(G''_r, \_)) &= \{q \to r : \&\mathbf{quit}\}
\end{aligned}$$

As an example of synchronization set, we have:

$$\begin{aligned}
&\mathsf{sync}_{\{p\},\{q\}}(\mathsf{mcomm}_{\{p\},\{q\}})(G'_p, G'_r) \\
&\quad = \{(p \to q : \&\mathbf{t}, 1, \\
&\quad (p \to r : \langle \mathrm{end} \rangle;\ \mathrm{close}), G'_r), \\
&\quad (1, q \to r : \&\mathbf{t}, G'_p, \\
&\quad (p \to r : \langle \mathrm{close} \rangle;\ \mathrm{close}))\}
\end{aligned}$$

We have that:

$$\begin{aligned}
G'_p \ {}^{p}\!\vee^{r}\ G'_r &= (p \to q : \&\mathbf{t};\ q \to r : \mathbf{ok};\ p \to r : \langle \mathrm{end} \rangle;\ \mathrm{close})\& \\
&\qquad (q \to r : \mathbf{ok};\ p \to q : \&\mathbf{t};\ p \to r : \langle \mathrm{end} \rangle;\ \mathrm{close}) \\
G'_p \ {}^{p}\!\vee^{r}\ G''_r &= (p \to q : \&\mathbf{t};\ q \to r : \&\mathbf{quit};\ \omega)\& \\
&\qquad (q \to r : \&\mathbf{quit};\ p \to q : \&\mathbf{t};\ \omega) \\
G''_p \ {}^{p}\!\vee^{r}\ G'_r &= (p \to q : \&\mathbf{f};\ q \to r : \mathbf{ok};\ \omega)\& \\
&\qquad (q \to r : \mathbf{ok};\ p \to q : \&\mathbf{f};\ \omega) \\
G''_p \ {}^{p}\!\vee^{r}\ G''_r &= (p \to q : \&\mathbf{f};\ q \to r : \&\mathbf{quit};\ \mathrm{close})\& \\
&\qquad (q \to r : \&\mathbf{quit};\ p \to q : \&\mathbf{f};\ \mathrm{close})
\end{aligned}$$

and finally

$$\begin{aligned}
G_p \ {}^{p}\!\vee^{r}\ G_r &= ((G'_p \ {}^{p}\!\vee^{r}\ G'_r) \,\&\, (G'_p \ {}^{p}\!\vee^{r}\ G''_r))\oplus \\
&\qquad ((G''_p \ {}^{p}\!\vee^{r}\ G'_r) \,\&\, (G''_p \ {}^{p}\!\vee^{r}\ G''_r)) \\
&\simeq_{\{p,r\}} (p \to q : \&\mathbf{t};\ q \to r : \mathbf{ok};\ p \to r : \langle \mathrm{end} \rangle;\ \mathrm{close})\oplus \\
&\qquad (p \to q : \&\mathbf{f};\ q \to r : \&\mathbf{quit};\ \mathrm{close})
\end{aligned}$$

Example 6.2 Continuing Example 4.2, we recall the types of the participants b_1, b_2, and s:

$$\begin{aligned}
G_1 &:= b_2 \to s : \mathbf{ok};\ b_2 \to s : \mathrm{string};\ s \to b_2 : \mathrm{date};\ \mathrm{close} \\
G_2 &:= b_2 \to s : \mathbf{quit};\ \mathrm{close}
\end{aligned}$$

$$G_{b_1} := b_1 \to s : \text{string}; s \to b_1 : \text{int}; b_1 \to b_2 : \text{int}; \text{close}$$
$$G_{b_2} := s \to b_2 : \text{int}; b_1 \to b_2 : \text{int}; (G_1 \oplus G_2)$$
$$G_s := b_1 \to s : \text{string}; s \to b_1, b_2 : \text{int}; (G_1 \,\&\, G_2)$$

Also, we pose $G := b_1 \to s : \text{string}; s \to b_1, b_2 : \text{int}; b_1 \to b_2 : \text{int}; (G_1 \oplus G_2)$. We have the following mergings:

$$G_{b_1} \;{}^{b_1}\!\vee^{b_2}\; G_{b_2} = (b_1 \to s : \text{string}; s \to b_1 : \text{int}; s \to b_2 : \text{int}; b_1 \to b_2 : \text{int}; (G_1 \oplus G_2)) \,\&\, G$$
$$G_{b_1} \;{}^{b_1}\!\vee^{s}\; G_s = b_1 \to s : \text{string}; s \to b_1, b_2 : \text{int}; b_1 \to b_2 : \text{int}; (G_1 \,\&\, G_2)$$
$$G_{b_2} \;{}^{b_1}\!\vee^{s}\; G_s = G$$

As we can see, $G_{b_1} \;{}^{b_1}\!\vee^{b_2}\; G_{b_2}$ considers two different cases: the case where s decides to send the quote separately, and the case where s sends the quote to both of them at once. Things get easier when we know the viewpoint of s. In the end, we have the following judgment:

$$\varnothing; P_{b_1} \big| P_{b_2} \big| P_s \vdash x : \langle G \big| b_1, b_2, s \rangle$$

Example 6.3 (Philosopher's dinner) Let us consider three philosophers p, q, r passing around one stick. Their processes can be defined as follows:

$$P_p := x^{pr}(n : \text{int}).\overline{x}^{pq}\langle n \rangle.\text{close}(x)$$
$$P_q := x^{qp}(n : \text{int}).\overline{x}^{qr}\langle n \rangle.\text{close}(x)$$
$$P_r := x^{rq}(n : \text{int}).\overline{x}^{rp}\langle n \rangle.\text{close}(x)$$

We can check that each process behaves well, according to the expected types:

$$\varnothing; P_p \vdash x : \langle r \to p : \text{int}; p \to q : \text{int}; \text{close} \big| p \rangle$$
$$\varnothing; P_q \vdash x : \langle p \to q : \text{int}; q \to r : \text{int}; \text{close} \big| q \rangle$$
$$\varnothing; P_r \vdash x : \langle q \to r : \text{int}; r \to p : \text{int}; \text{close} \big| r \rangle$$

Also, every pair of process behaves well, as we can type them in our typing system:

$$\varnothing; P_p \big|_x P_q \vdash x : \langle r \to p : \text{int}; p \to q : \text{int}; q \to r : \text{int}; \text{close} \big| p, q \rangle$$
$$\varnothing; P_p \big|_x P_r \vdash x : \langle q \to r : \text{int}; r \to p : \text{int}; p \to q : \text{int}; \text{close} \big| p, r \rangle$$
$$\varnothing; P_q \big|_x P_r \vdash x : \langle p \to q : \text{int}; q \to r : \text{int}; r \to p : \text{int}; \text{close} \big| q, r \rangle$$

However, the three processes together can deadlock, as it is witnessed by the merging operation; in fact

$$((r \to p : \text{int}; p \to q : \text{int}; \text{close}) \;{}^{p}\!\vee^{q}\; (p \to q : \text{int}; q \to r : \text{int}; \text{close})) \;{}^{p,q}\!\vee^{r}\; (q \to r : \text{int}; r \to p : \text{int}; \text{close}) = \omega$$

and hence $P_p \big|_x P_q \big|_x P_r$ cannot be typed.

Example 6.4 Let us see an example of three processes involved in a multicast communication with a subsession:

$$P_p := \overline{x}^{pqr}(y).\overline{x}^{pq}(z).\text{wait}(y).\text{wait}(z).\text{close}(x)$$
$$P_q := x^{qp}(y).(\text{close}(y) \parallel x^{qp}(z).(\text{close}(z) \parallel \text{close}(x)))$$
$$P_r := x^{rp}(y).(\text{close}(y) \parallel \text{close}(x))$$

We pose the following four types:

$$G_1 := p \to qr : \langle \text{end} \rangle; p \to q : \langle \text{end} \rangle; \text{close}$$
$$G_1' := p \to qr : \langle \text{close} \rangle; p \to q : \langle \text{close} \rangle; \text{close}$$
$$G_2 := p \to q : \langle \text{close} \rangle; p \to qr : \langle \text{close} \rangle; \text{close}$$
$$G_3 := p \to q : \langle \text{close} \rangle; p \to q : \langle \text{close} \rangle; p \to r : \langle \text{close} \rangle; \text{close}$$

Note that $G_1' \not\simeq_{\{q,r\}} G_2$, and that G_3 can be reorder under $\simeq_{\{q,r\}}$:

$$G_3 \simeq_{\{q,r\}} p \to r : \langle \text{close} \rangle; p \to q : \langle \text{close} \rangle; p \to q : \langle \text{close} \rangle; \text{close}$$

We can prove that the following typing judgments hold:

$$\varnothing; P_p \vdash x : \langle G_1 \big| p \rangle$$
$$\varnothing; P_q \vdash x : \langle p \to q : \langle \text{close} \rangle; p \to q : \langle \text{close} \rangle; \text{close} \big| q \rangle$$
$$\varnothing; P_r \vdash x : \langle p \to r : \langle \text{close} \rangle; \text{close} \big| r \rangle$$
$$\varnothing; P_p \big|_x P_q \vdash x : \langle G_1 \big| p, q \rangle$$
$$\varnothing; P_p \big|_x P_r \vdash x : \langle G_1 \big| p, r \rangle$$
$$\varnothing; P_q \big|_x P_r \vdash x : \langle G_1' \,\&\, G_2 \,\&\, G_3 \big| q, r \rangle$$
$$\varnothing; P_p \big|_x P_q \big|_x P_r \vdash x : \langle G_1 \big| p, q, r \rangle$$

As we can see, P_p is the process that imposes some particular synchronized scheduling for everyone. With the viewpoint of participants q and r, but without p, we cannot know if session x should follow G_1', G_2, or G_3, hence the type $G_1' \,\&\, G_2 \,\&\, G_3$ for $P_q \big|_x P_r$. If we merge these session types with G_1, we get three possible behaviors, two of which are not feasible:

$$G_1 \;{}^{p}\!\vee^{q,r}\; G_1' \simeq_{\{p,q,r\}} G_1 \qquad G_1 \;{}^{p}\!\vee^{q,r}\; G_2 \simeq_{\{p,q,r\}} \omega$$
$$G_1 \;{}^{p}\!\vee^{q,r}\; G_3 \simeq_{\{p,q,r\}} \omega$$

As a consequence, $G_1 \;{}^{p}\!\vee^{q,r}\; (G_1 \,\&\, G_2 \,\&\, G_3) \simeq_{\{p,q,r\}} G_1$, which is the type we get for x when typing $P_p \big|_x (P_q \big|_x P_r)$. We would get the same type (modulo equivalence) by combining these three processes in any different order.

7 A semantic interpretation of partial session types and the merge operator

In this section, we provide a semantic interpretation of session types and their operations, in particular the merge function defined above. Essentially, the type of a session tells us what happens in the session from a particular *viewpoint*, and merging session types requires us to merge interpretations from different viewpoints. To this end, we need to define suitable categories and constructions taking viewpoints into account. Let us summarize this section briefly.

First, in Sect. 7.1 we introduce the category of *communication structures*, whose objects are sets of events (i.e., the basic communication steps) endowed with an *independence* relation; if two events are independent from a given viewpoint, the observer cannot tell the actual order of their execution. Maps between communication structures allow to move from one viewpoint to another.

Given a communication structure, we define *schedulable sets* as sets of traces taken up-to the equivalence induced by the independence relation (Sect. 7.2). Equivalent traces denote executions with different interleavings which cannot be distinguished from a given viewpoint. Intuitively, a chain of communications $G = C_1; \ldots ; C_n$ is interpreted as an equivalence class of traces $[G]_{\simeq_S}$, where the equivalence relation $\simeq_S$ depends on the viewpoint S. Similarly, a conjunction of communication chains is interpreted as a schedulable set, i.e., a union of these equivalence classes. Thus, a schedulable set can be seen as a deterministic strategy represented by traces (up-to equivalence) generated by interacting with any possible scheduler. Finally, we introduce *trace sets*, which are collections of schedulable sets; trace sets are used to interpret the non-deterministic behavior given by internal choices in the types. Therefore, we anticipate all possible internal choices by choosing a schedulable set at the beginning, and then following deterministically the traces therein, depending on specific choices of the scheduler. This intuitive interpretation of types is formalized in Sect. 7.3. Leveraging these constructions, in Sect. 7.4 we give the interpretation of the merge operator as the structure of a lax monoidal functor from the category of viewpoints to that of communication structures; this provides important properties of the operator itself, such as associativity and stability under $\simeq_S$.

7.1 Communication structures

We start with the fundamental elements of session types, i.e., communications.

Definition 7.1 (Communication structure) A *communication structure* A is a triple $A = (E_A, I_A, 1_A)$ where E_A is a set, $1_A \in E_A$, $I_A \subseteq E_A \times E_A$ is a symmetric relation called the *independence relation* such that $\forall x, x\ I_A\ 1_A$.

Intuitively, 1_A denotes a no-operation, or rather, a *no-communication*.

Definition 7.2 (Category COMM) We define COMM as a category where:

- an object is a communication structure
- a morphism $f : (E_A, I_A, 1_A) \to (E_B, I_B, 1_B)$ is a partial function from E_A to E_B such that $f(1_A) = 1_B$, and, for any $x, y \in E_A$ such that both $f(x)$ and $f(y)$ are defined, we have that $f(x)\ I_B\ f(y)$ iff $x\ I_A\ y$;
- composition and identities are standard.

Intuitively, an object in the category can define all possible communications from some viewpoint, and morphisms allow us to change the viewpoint.

Definition 7.3 (Monoidal product) The monoidal product of $(E_A, I_A, 1_A)$ and $(E_B, I_B, 1_B)$ is $(E_C, I_C, 1_C) = (E_A, I_A, 1_A) \otimes (E_B, I_B, 1_B)$ defined as follows:

- $E_C = E_A \times E_B$, with projections $\pi_A : E_A \times E_B \to E_A$, $\pi_B : E_A \times E_B \to E_B$;
- for all $x_A, y_A \in E_A$ and $x_B, y_B \in E_B$, $(x_A, x_B)\ I_C\ (y_A, y_B) \Leftrightarrow x_A\ I_A\ y_A$ and $x_B\ I_B\ y_B$;
- $1_C = (1_A, 1_B)$

The unit of this product is $J := (\{1_J\}, \{(1_J, 1_J)\}, 1_J)$. The natural isomorphisms $\alpha_{A,B,C} : A \otimes (B \otimes C) \simeq (A \otimes B) \otimes C$, $\lambda_A : J \otimes A \simeq A$, and $\rho_A : A \otimes J \simeq A$ are defined by $\alpha_{A,B,C}(x, (y, z)) := ((x, y), z)$, $\lambda_A(1_J, x) := x$, and $\rho_A(x, 1_J) := x$.

Proposition 7.1 $(\text{COMM}, \otimes, J, \alpha, \lambda, \rho)$ *is a symmetric monoidal category.*

Notice that this product is not cartesian, because projections are not morphisms.

The viewpoint of a session is the set of local participants of this session.

Definition 7.4 (Category VIEW) The thin category VIEW is the partial order $(\wp(\mathfrak{P}), \subseteq)$ viewed as a strict symmetrical monoidal category, i.e.,

- objects are subsets of the set of participants $\mathfrak{P}$
- there exists a unique morphism $f : A \to B$ iff $A \subseteq B$
- the tensor $A \otimes B$ is the union $A \cup B$
- the unit is $\varnothing$.

A functor from VIEW to COMM would allow us to consider communication structures from different viewpoints, and to merge a pair of compatible communications from different viewpoints into a communication from a unified viewpoint. The functor must preserve the monoidal structure, and moreover, the merge morphism given by this functor is a partial function because incompatible communications are impossible to merge. The right notion for this is that of *lax symmetric monoidal functor*, which we recall next.

Definition 7.5 A *lax symmetric monoidal functor* (L, μ, υ) from $(\text{View}, \cup, \varnothing)$ to $(\text{Comm}, \otimes, J, \alpha, \lambda, \rho)$, consists of a functor $L : \text{View} \to \text{Comm}$, a natural transformation $\mu_{S_1,S_2} : L(S_1) \otimes L(S_2) \to L(S_1 \cup S_2)$ and a morphism $\upsilon : J \to L(\varnothing)$, such that the following diagrams commute:

$$\begin{tikzcd}
(L(S_1) \otimes L(S_2)) \otimes L(S_3) \arrow[r, "\alpha"] \arrow[d, "\mu_{S_1,S_2}\otimes \mathrm{id}"] & L(S_1) \otimes (L(S_2) \otimes L(S_3)) \arrow[d, "\mathrm{id}\otimes\mu_{S_2,S_3}"] \\
L(S_1 \cup S_2) \otimes L(S_3) \arrow[d, "\mu_{S_1\cup S_2,S_3}"] & L(S_1) \otimes L(S_2 \cup S_3) \arrow[d, "\mu_{S_1,S_2\cup S_3}"] \\
L(S_1 \cup S_2 \cup S_3) \arrow[r, leftrightarrow, "\mathrm{id}"] & L(S_1 \cup S_2 \cup S_3)
\end{tikzcd}$$

$$\begin{tikzcd}
L(S_1) \otimes L(S_2) \arrow[r, "\gamma"] \arrow[d, "\mu_{S_1,S_2}"] & L(S_2) \otimes L(S_1) \arrow[d, "\mu_{S_2,S_1}"] \\
L(S_1 \cup S_2) \arrow[r, leftrightarrow, "\mathrm{id}"] & L(S_2 \cup S_1)
\end{tikzcd}$$

$$\begin{tikzcd}
L(S) \otimes J \arrow[r, "\mathrm{id}\otimes\upsilon"] \arrow[d, "\rho"] & L(S) \otimes L(\varnothing) \arrow[d, "\mu_{A,\varnothing}"] & J \otimes L(S) \arrow[r, "\upsilon\otimes\mathrm{id}"] \arrow[d, "\lambda"] & L(\varnothing) \otimes L(S) \arrow[d, "\mu_{\varnothing,A}"] \\
L(S) \arrow[r, leftrightarrow, "\mathrm{id}"] & L(S) & L(S) \arrow[r, leftrightarrow, "\mathrm{id}"] & L(S)
\end{tikzcd}$$

7.2 Traces

Let $A = (E_A, I_A, 1_A)$ be a communication structure. We note by E_A^* the set of finite sequences over E_A, and Set_* for the category of sets and partial functions.

Definition 7.6 (Functor $\hat{\_}$) We define $\hat{\_} : \text{Comm} \to \text{Set}_*$ as follows:

$$\hat{A} := E_A^*$$

$$\hat{f}(\epsilon) := \epsilon \qquad \hat{f}(as) := \begin{cases} f(a)\hat{f}(s) & \text{if } f(a) \text{ and } \hat{f}(s) \text{ are defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

A *trace (over A)* is an element $\langle a_1 \dots a_n\rangle \in E_A^*$. We only consider traces where no error occurs, thus if a morphism f is undefined at some point in the trace s, there is an error, and as a consequence $\hat{f}(s)$ is undefined.

Definition 7.7 (Equivalence relation) For any communication structure A, we define $\simeq_A$ the smallest equivalence relation on E_A^* such that $sabt \simeq_A sbat$ if $a\ I_A\ b$, and $s1t \simeq_A st$.

Definition 7.8 (Schedulable and trace sets) A *schedulable set (over A)* is a set $C \subseteq E_A^*$ closed under $\simeq_A$, that is, for all $s, t \in E_A^*$, if $s \in C$ and $s \simeq_A t$ then $t \in C$.

A *trace set (over A)* is a set B of schedulable sets.

Intuitively, a trace set B over A denotes a process which selects a schedulable set $C \in B$ and proposes it to a scheduler. The scheduler then chooses a trace $s \in C$ and s is executed. C is closed under $\simeq_A$, because the interleaving of the events is chosen by the scheduler.

Definition 7.9 (Functor $\mathcal{T}$) For any communication structure A, let us denote $\mathcal{T}(A)$ the set of all trace sets over A. For any communication morphism $f : A_1 \to A_2$, we define $\mathcal{T}(f) : \mathcal{T}(A_1) \to \mathcal{T}(A_2)$ by

$$\mathcal{T}(f)(B) = \Big\{\{\hat{f}(w) \big| w \in C, \hat{f}(w) \text{ is defined}\} \big| C \in B\Big\}$$

This defines a functor $\mathcal{T} : \text{Comm} \to \text{Set}$, which we call the trace set functor.

Definition 7.10 (Product of trace sets) If $B_1 \in \mathcal{T}(A_1)$ and $B_2 \in \mathcal{T}(A_2)$, we define $B_1 \otimes B_2 \in \mathcal{T}(A_1 \otimes A_2)$ as the following set of sets:

$$\Big\{\big\{\langle(a_1, b_1) \dots (a_n, b_n)\rangle \big| \langle a_1 \dots a_n\rangle \in C_1, \langle b_1 \dots b_n\rangle \in C_2\big\} \\ \big| C_1 \in B_1, C_2 \in B_2\Big\}$$

Note that, if $\varnothing \in B_1$ or $\varnothing \in B_2$, then $\varnothing \in B_1 \otimes B_2$.

Informally, $B_1 \otimes B_2$ corresponds to the synchronized threading of two processes, where a communication (a_i, b_i) should be a pair containing the same communication from two different viewpoints. If it is not the case, then the trace containing (a_i, b_i) is an incorrect synchronization, therefore a merger morphism should not be defined over it.

Definition 7.11 We define the following operations:

- We note $[w]_A$ for the equivalence class of w modulo $\simeq_A$. We note $[w]$ if A is clear from the context
- We define the commutative and associative operator on trace sets $B_1 \Cup B_2 := \{C_1 \cup C_2 \big| C_1 \in B_1, C_2 \in B_2\}$. In particular, $B \Cup \varnothing = \varnothing$
- We also define $\Cup\{B_1, \dots, B_n\} := B_1 \Cup \dots \Cup B_n$, with the special case $\Cup\varnothing := \{\varnothing\}$, because $\{\varnothing\}$ is the neutral element for $\Cup$
- We define $a \cdot_A B := \{\bigcup_{s\in C}[as]_A \big| C \in B\}$.

Example 7.1 Let us see an example of the tensor product $\otimes$. Let $A_1 = (\{a, b\}, I_{A_1}, 1)$, $A_2 = (\{a, c\}, I_{A_2}, 1)$ with I_{A_1} and I_{A_2} being the smallest independence relations possible; e.g., $I_{A_1} = \{(a, 1), (b, 1), (1, a), (1, b), (1, 1)\}$. Then

$$\begin{aligned}\{[ab]_{A_1}\} \otimes \{[ac]_{A_2}\} = &\{[(a, 1)(b, 1)(1, a)(1, c)]\} \Cup \{[(a, 1)(b, a)(1, c)]\} \\ &\Cup \{[(a, 1)(1, a)(b, c)]\} \Cup \{[(a, a)(b, 1)(1, c)]\} \\ &\Cup \{[(a, a)(b, c)]\} \Cup \{[(1, a)(a, c)(b, 1)]\}\end{aligned}$$

The elements of these equivalence classes are all possible linearizations of six “happens-before” orders, which can be represented graphically as follows:

$$\begin{tikzcd}(a,1) \arrow[d] & (1,a) \arrow[d] \\ (b,1) & (1,c)\end{tikzcd}
\quad
\begin{tikzcd}(a,1) \arrow[d] \\ (b,a) \arrow[d] \\ (1,c)\end{tikzcd}
\quad
\begin{tikzcd}(a,1) \arrow[d] & (1,a) \arrow[dl] \\ (b,c) &\end{tikzcd}
\quad
\begin{tikzcd}(a,a) \arrow[d] \arrow[dr] & \\ (b,1) & (1,c)\end{tikzcd}
\quad
\begin{tikzcd}(a,a) \arrow[d] \\ (b,c)\end{tikzcd}
\quad
\begin{tikzcd}(1,a) \arrow[d] \\ (a,c) \arrow[d] \\ (b,1)\end{tikzcd}$$

Let $A_3 = (\{a, b, c\}, I_{A_3}, 1)$, with I_{A_3} being the smallest independence relation such that $b \ I_{A_3} \ c$. Let us choose a merge function $f : A_1 \otimes A_2 \to A_3$ defined as $f(a, a) = a$, $f(1, c) = c$ and $f(b, 1) = b$; undefined otherwise. Then, we have that

$$\mathcal{T}(f)(\{[ab]_{A_1}\} \otimes \{[ac]_{A_2}\}) = \{[abc]_{A_3}\}$$

which corresponds to
$$\begin{tikzcd}a \arrow[d] \arrow[dr] & \\ b & c\end{tikzcd}$$

since only in the fourth equivalence class the function f is defined for all communications.

We conclude this section with some useful results about trace sets.

Lemma 7.1 *We have the following properties about the product of trace sets:*

$$(B_1 \cup B_1') \otimes B_2 = (B_1 \otimes B_2) \cup (B_1' \otimes B_2)$$
$$\{C_1 \cup C_1'\} \otimes \{C_2\} = (\{C_1\} \otimes \{C_2\}) \uplus (\{C_1'\} \otimes \{C_2\})$$
$$\{[\epsilon]_{A_1}\} \otimes \{[\epsilon]_{A_2}\} = \{[\epsilon]_{A_1 \otimes A_2}\}$$
$$(a \cdot_{A_1} \{[s]_{A_1}\}) \otimes \{[\epsilon]_{A_2}\} = (a, 1_{A_2}) \cdot_{A_1 \otimes A_2} (\{[s]_{A_1}\} \otimes \{[\epsilon]_{A_2}\})$$
$$\{[\epsilon]_{A_1}\} \otimes (a \cdot_{A_2} \{[s]_{A_2}\}) = (1_{A_1}, a) \cdot_{A_1 \otimes A_2} (\{[\epsilon]_{A_1}\} \otimes \{[s]_{A_2}\})$$

and, if $\{[\epsilon]_{A_1}\} \not\subseteq C_1$ *and* $\{[\epsilon]_{A_2}\} \not\subseteq C_2$*:*

$$\{C_1\} \otimes \{C_2\} = \uplus\{(a, b) \cdot_{A_1 \otimes A_2} (\{[s]_{A_1}\} \otimes \{[t]_{A_2}\}) \big| as \in C_1, bt \in C_2, (a, b) \neq (1_{A_1}, 1_{A_2})\}$$

Proof By double inclusion. □

Lemma 7.2 *If $f : A_1 \to A_2$, and $B, B' \in \mathcal{T}(A_1)$ then we have the following:*

- $\mathcal{T}(f)(B \cup B') = \mathcal{T}(f)(B) \cup \mathcal{T}(f)(B')$
- $\mathcal{T}(f)(B \uplus B') = \mathcal{T}(f)(B) \uplus \mathcal{T}(f)(B')$
- $\mathcal{T}(f)(\{[\epsilon]_{A_1}\}) = \{[\epsilon]_{A_1}\}$
- *if $f(a)$ is defined, then* $\mathcal{T}(f)(a \cdot_{A_1} B) = f(a) \cdot_{A_2} \mathcal{T}(f)(B)$
- $\mathcal{T}(f)(a \cdot_{A_1} \varnothing) = \varnothing$
- *if $B \neq \varnothing$ and $f(a)$ is not defined, then* $\mathcal{T}(f)(a \cdot_{A_1} B) = \{\varnothing\}$.

Proof By double inclusion. □

Lemma 7.3 *Let* $A_1, A_2, A_3 \in \textsc{Comm}$, $f : A_1 \otimes A_2 \to A_3$, *and* $a_1 \in A_1$ *such that for any* $a_2 \in A_2$, $f(a_1, a_2)$ *is defined iff* $a_2 = 1$. *Then, for any* $B_1 \in \mathcal{T}(A_1)$, $B_2 \in \mathcal{T}(A_2)$, *we have that* $\mathcal{T}(f)((a_1 \cdot_{A_1} B_1) \otimes B_2) = f(a_1, 1) \cdot_{A_1 \otimes A_2} \mathcal{T}(f)(B_1 \otimes B_2)$.

Proof Using Lemmas 7.1 and 7.2, we proceed by Noetherian induction on the sum of the size of B_1 and B_2. The base case is when $B_1 = \{C_1\}$ and $B_2 = \{C_2\}$, then we proceed by Noetherian induction on the sum of the size of C_1 and C_2. The base case is when $C_1 = [w_1]_{A_1}$ and $C_2 = [w_2]_{A_2}$, then we proceed by Noetherian induction on the sum of the length of w_1 and w_2. □

Lemma 7.4 *Let* $A_1, A_2, A_3 \in \textsc{Comm}$, $f : A_1 \otimes A_2 \to A_3$, $a_1 \in A_1$, *and* $a_2 \in A_2$ *such that* $f(a_1, a_2)$ *is defined and:*

$$\forall a \in A_2, a \ I_{A_2} \ a_2 \Rightarrow f(a_1, a) \text{ is undefined}$$
$$\forall a \in A_1, a \ I_{A_1} \ a_1 \Rightarrow f(a, a_2) \text{ is undefined}$$

Then, for any $B_1 \in \mathcal{T}(A_1)$, $B_2 \in \mathcal{T}(A_2)$, *we have that:*

$$\mathcal{T}(f)((a_1 \cdot_{A_1} B_1) \otimes (a_2 \cdot_{A_2} B_2)) = f(a_1, a_2) \cdot_{A_1 \otimes A_2} \mathcal{T}(f)(B_1 \otimes B_2)$$

Proof The proof is done the same way as in Lemma 7.3. □

7.3 Interpreting session types as trace sets

Before defining the interpretation of session types, we need to consider types where every communication involves at least one participant of a given viewpoint, because those are the communications we can observe from the behavior of internal participants of a session. Also, the special case of a process with no participant will be seen as neutral for composition, so it can only close.

Definition 7.12 (Engaged communication and session type) A communication $C \in \mathfrak{C}$ is *engaged on* S if C is terminal, or if $C = 1$, or if $C = p' \to \tilde{q} : m$ and $S \cap (\{p'\} \cup \tilde{q}) \neq \varnothing$. If $S = \varnothing$, we require also that C is either close or 1.

A session type is *engaged on* S when every communication it contains is engaged on S.

We denote by $\mathfrak{C}_S$ and $\mathfrak{G}_S$ the sets of communications and types engaged on S, respectively.

For a viewpoint S, we can consider the structure of all communications which are visible (i.e., engaged) from that viewpoint.

Definition 7.13 We define a functor $L : \textsc{View} \to \textsc{Comm}$ as follows. If $S = \varnothing$, then $L(S) := J$. Otherwise, $L(S) := ((\mathfrak{C}_S - \{\text{close}\})/{\simeq_S}, I_S, 1)$, where the quotient over communications is given by the equivalence $\simeq_S$ defined as

$$p \to \tilde{q} : \langle G_1 \rangle \simeq_S p \to \tilde{q} : \langle G_2 \rangle \iff G_1 \simeq_S G_2.$$

Definition 7.14 (Interpretation of session types) Let S be a viewpoint. We define an interpretation function $[\![\_]\!]_S : \mathfrak{G}_S \to \mathcal{T}(L(S))$ for session types engaged on S, as follows:

$$\begin{array}{rlrl}
[\![0]\!]_S &:= \varnothing & [\![\omega]\!]_S &:= \{\varnothing\} \\
[\![G_1 \mathbin{\&} G_2]\!]_S &:= [\![G_1]\!]_S \uplus [\![G_2]\!]_S & [\![G_1 \oplus G_2]\!]_S &:= [\![G_1]\!]_S \cup [\![G_2]\!]_S \\
[\![C; G]\!]_S &:= C \cdot_{L(S)} [\![G]\!]_S & [\![\text{close}]\!]_S &:= \{[\epsilon]_{L(S)}\} \\
[\![C]\!]_S &:= \{[C]_{L(S)}\} \quad \text{otherwise.} & &
\end{array}$$

Lemma 7.5 *If $G_1 \simeq_S G_2$, then $[\![G_1]\!]_S = [\![G_2]\!]_S$.*

Proof By induction on $G_1 \simeq_S G_2$. □

Lemma 7.6 *If G is a chain $C_1; \ldots; C_n$, then $[\![G]\!]_S = \{[C_1; \ldots; C_n]_{L(S)}\}$.*

Proof By induction on n. □

Lemma 7.7 *For any session type $G = \&\{G_1, \ldots, G_n\}$, where all the G_i are chains of communications, we have that:*

$$[\![G]\!]_S = \{\bigcup_{i=1}^{n} [G_i]_{L(S)}\}$$

Proof By induction on n. □

Lemma 7.8 *For any session type in DNF $G = \bigoplus \{\& A_1, \ldots, \& A_n\}$, we have*

$$[\![G]\!]_S = \bigcup_{i=1}^{n} [\![\& A_i]\!]_S$$

Proof By induction on n. □

Lemma 7.9 *If $[\![G_1]\!]_S = [\![G_2]\!]_S$, then $G_1 \simeq_S G_2$.*

Proof Thanks to Lemma 7.5, we can suppose that G_1 and G_2 are in DNF.

- If both G_1 and G_2 are chains of communications, then, using Lemma 7.6, we have that $[G_1]_{L(S)} = [G_2]_{L(S)}$, therefore $G_1 \simeq_S G_2$.
- If both G_1 (respectively, G_2) is of the form $\& A_1$ (respectively, $\& A_2$), where A_1 and A_2 are finite sets of chains of communications, then, using Lemma 7.7, we have that $\{\bigcup_{G' \in A_1} [G']_{L(S)}\} = \{\bigcup_{G'' \in A_2} [G'']_{L(S)}\}$. We can conclude by noting that equivalence classes are distinct and using point 1.
- If both G_1 (respectively, G_2) is of the form $\bigoplus\{\& A_1, \ldots, \& A_n\}$ (respectively, $\bigoplus\{\& A'_1, \ldots, \& A'_n\}$), where the A_i and A'_i are finite sets of chains of communications, then, using Lemma 7.8, we have that $\bigcup_{i=1}^{n} [\![\& A_i]\!]_S = \bigcup_{i=1}^{n} [\![\& A'_i]\!]_S$. We can conclude by noting that each $[\![\& A_i]\!]_S$ and each $[\![\& A'_i]\!]_S$ are singletons, and using point 2. □

We now can state that our semantics is sound and complete with respect to the relation $\simeq_S$.

Theorem 7.1 *$G_1 \simeq_S G_2$ iff $[\![G_1]\!]_S = [\![G_2]\!]_S$.*

Proof By Lemmas 7.5 and 7.9. □

7.4 Interpretation of the merge operator

We conclude this section with some useful properties about merge functions and in particular the merge operator ${}^{S_1}\vee^{S_2}$.

Lemma 7.10 *Let $f : L(S_1) \otimes L(S_2) \to L(S_1 \cup S_2)$ be a morphism in* Comm. *If $\mathsf{map}_{S_1,S_2}(f)(G_1, G_2)$ is defined, then we have:*

$$[\![\mathsf{map}_{S_1,S_2}(f)(G_1, G_2)]\!]_{S_1 \cup S_2} = \mathcal{T}(f)([\![G_1]\!]_{S_1} \otimes [\![G_2]\!]_{S_2})$$

Proof By syntactic induction on (G_1, G_2), using Lemmas 7.1 and 7.2. □

Lemma 7.11 *$(L, \mathsf{mcomm}, id_J)$ is a lax symmetric monoidal functor.*

Proof Properties for the unitors are trivial. Associativity for C_1, C_2, C_3 is done by strong recursion on the measure $h(C_1) + h(C_2) + h(C_3)$. Commutativity for C_1, C_2 is done by induction on the measure $h(C_1) + h(C_2)$. The recursive case happens when we need to prove associativity, i.e.,

$$\begin{array}{c}
\mathsf{map}_{S_1,S_2\cup S_3}(\mathsf{mcomm}_{S_1,S_2\cup S_3})(G_1, \mathsf{map}_{S_2,S_3}(\mathsf{mcomm}_{S_2,S_3})(G_2, G_3)) \\
\simeq_{S_1\cup S_2\cup S_3} \\
\mathsf{map}_{S_1\cup S_2,S_3}(\mathsf{mcomm}_{S_1\cup S_2,S_3})(\mathsf{map}_{S_1,S_2}(\mathsf{mcomm}_{S_1,S_2})(G_1, G_2), G_3)
\end{array}$$

and symmetry, i.e.,

$$\mathsf{map}_{S_1,S_2}(\mathsf{mcomm}_{S_1,S_2})(G_1, G_2) \simeq_{S_1\cup S_2} \mathsf{map}_{S_2,S_1}(\mathsf{mcomm}_{S_2,S_1})(G_2, G_1)$$

By recursion, we know that mcomm is associative and commutative for communications with a strictly smaller height, therefore we can conclude. □

Now we can state several important properties of the merge function.

Theorem 7.2 *All the possible values of $G_1 \;{}^{S_1}\vee^{S_2}\; G_2$ are $\simeq_{S_1\cup S_2}$-equivalent. Moreover, if $G'_1 \simeq_{S_1} G_1$ and $G'_2 \simeq_{S_2} G_2$, then $G_1 \;{}^{S_1}\vee^{S_2}\; G_2 \simeq_{S_1\cup S_2} G'_1 \;{}^{S_1}\vee^{S_2}\; G'_2$.*

Proof By Lemmas 7.10 and 7.11, we have that $\mathsf{map}_{S_1,S_2}(\mathsf{mcomm}_{S_1,S_2})$ corresponds to a morphism in Set$(\mathcal{T}(L(S_1) \otimes L(S_2)), \mathcal{T}(L(S_1 \cup S_2)))$, hence we can conclude using Theorem 7.1. □

Thanks to the previous theorem, we can consider that $G_1 \;{}^{S_1}\vee^{S_2}\; G_2$ is defined for any G_1 and G_2 by choosing any DNF of G_1 and G_2.

Theorem 7.3 *We have that:*

$$G_1 \;{}^{S_1}\!\vee^{S_2}\; G_2 \simeq_{S_1 \cup S_2} G_2 \;{}^{S_2}\!\vee^{S_1}\; G_1$$

$$G_1 \;{}^{S_1}\!\vee^{S_2 \cup S_3}\; (G_2 \;{}^{S_2}\!\vee^{S_3}\; G_3) \simeq_{S_1 \cup S_2 \cup S_3} (G_1 \;{}^{S_1}\!\vee^{S_2}\; G_2) \;{}^{S_1 \cup S_2}\!\vee^{S_3}\; G_3$$

We conclude with two technical lemmas that will be useful in the following.

Lemma 7.12 *If* $\{p\} \cup \tilde{q} \subseteq S_1$, $S_1 \cap S_2 = \varnothing$, *and* G_2 *is engaged on* S_2, *then* $(p \to \tilde{q} : m; G_1) \;{}^{S_1}\!\vee^{S_2}\; G_2 \simeq_{S_1 \uplus S_2} p \to \tilde{q} : m; (G_1 \;{}^{S_1}\!\vee^{S_2}\; G_2)$.

Proof Easy, using Lemma 7.3. □

Lemma 7.13 *If* $p \in S_1$, $p \neq q$, $\tilde{q}' \subseteq \tilde{q}$, *and* G *is engaged on* $\{q\}$, *then:*

$$\begin{aligned}
p \to \tilde{q} : \langle G_1' \rangle; G_1 \;{}^{S_1}\!\vee^{\{q\}}\; p \to \tilde{q}' : \langle G_2' \rangle; G_2 &\simeq_{S_1 \cup \{q\}} \\
&\quad p \to \tilde{q} : \langle G_1' \;{}^{S_1}\!\vee^{\{q\}}\; G_2' \rangle; \\
&\quad (G_1 \;{}^{S_1}\!\vee^{\{q\}}\; G_2) \\
p \to \tilde{q} : \&l; G_1 \;{}^{S_1}\!\vee^{\{q\}}\; p \to \tilde{q}' : \&l; & \\
&\quad G_2 \simeq_{S_1 \cup \{q\}} p \to \tilde{q} : \&l; (G_1 \;{}^{S_1}\!\vee^{\{q\}}\; G_2) \\
p \to \tilde{q} : A; G_1 \;{}^{S_1}\!\vee^{\{q\}}\; p \to \tilde{q}' : A; & \\
&\quad G_2 \simeq_{S_1 \cup \{q\}} p \to \tilde{q} : A; (G_1 \;{}^{S_1}\!\vee^{\{q\}}\; G_2)
\end{aligned}$$

Proof Easy, using Lemma 7.4. □

8 Subject reduction and progress

In this section, we state two main properties of session types, *subject reduction* and *progress*, which guarantee that "well-typed systems cannot go wrong." To this end, we first define a reduction semantics for partial session types.

Definition 8.1 (Reductions for session types) Actions γ for session types are defined as

$$\gamma ::= \oplus \big| p \to \tilde{q} : \langle \cdot \rangle \big| p \to \tilde{q} : \&l \big| p \to \tilde{q} : A$$

We write $G_1 \xrightarrow{\gamma}_S G_2$ for a transition from G_1 to G_2 from the viewpoint of S under the action γ. This relation is defined as follows:

$$G_1 \oplus G_2 \xrightarrow{\oplus}_S G_i \qquad p \to \tilde{q} : \langle G_1 \rangle; G_2 \xrightarrow{p \to \tilde{q} : \langle \cdot \rangle}_S G_2$$

$$p \to \tilde{q} : \&l; G \xrightarrow{p \to \tilde{q} : \&l}_S G \qquad p \to \tilde{q} : A; G \xrightarrow{p \to \tilde{q} : A}_S G$$

$$\frac{G_1 \xrightarrow{\gamma}_S G' \quad G_1 \simeq_S G_2}{G_2 \xrightarrow{\gamma}_S G'}$$

Note that transitions are not deterministic, in particular $G \simeq_S G \oplus G$, therefore we always have $G \xrightarrow{\oplus}_S G$, which is useful in case we are reducing an internal choice which is irrelevant for G.

Definition 8.2 (Reduction for environments) The reduction relation for process typing environments is $\Gamma_1 \xrightarrow{\alpha} \Gamma_2$, labelled by actions for processes α (see Definition 3.3), and it is defined as follows:

$$\frac{}{\cdot \xrightarrow{\alpha} \cdot} \qquad \frac{}{\Gamma \xrightarrow{\tau} \Gamma} \qquad \frac{G_1 \xrightarrow{\oplus}_S G_2 \quad \Gamma_1 \xrightarrow{\oplus} \Gamma_2}{x : \langle G_1 \big| S \rangle, \Gamma_1 \xrightarrow{\oplus} x : \langle G_2 \big| S \rangle, \Gamma_2}$$

$$\frac{G_1 \xrightarrow{\gamma}_S G_2}{x : \langle G_1 \big| S \rangle, \Gamma \xrightarrow{x:\gamma} x : \langle G_2 \big| S \rangle, \Gamma}$$

Then, the type system enjoys the following important properties.

Lemma 8.1 *(Inversion lemma) We have the following properties:*

- *If* $E; \overline{x}^{p\tilde{q}}\langle M \rangle.P \vdash \Gamma$, *then* $\Gamma \simeq \Gamma', x : \langle G \big| \{p\} \cup S \rangle$, *with* $G = p \to \tilde{q} : A; G'$ *engaged on* $\{p\}$, $S \cap \mathrm{fn}(G) = \varnothing$, $E \vdash_D M : A$, *and* $E; P \vdash \Gamma', x : \langle G' \big| p \rangle$
- *If* $E; x^{pq}(y : A).P \vdash \Gamma$, *then* $\Gamma \simeq \Gamma', x : \langle G \big| \{p\} \cup S \rangle$, *with* $G = q \to \tilde{p} : A; G'$ *engaged on* $\{p\}$, $S \cap \mathrm{fn}(G) = \varnothing$, $E, y : A; P \vdash \Gamma', x : \langle G' \big| p \rangle$, *and* $p \in \tilde{p}$
- *If* $E; \overline{x}^{p\tilde{q}}(y).P \vdash \Gamma$, *then* $\Gamma \simeq \Gamma', x : \langle G \big| \{p\} \cup S \rangle$, *with* $G = p \to \tilde{q} : \langle G_1 \rangle; G_2$ *engaged on* $\{p\}$, $S \cap \mathrm{fn}(G) = \varnothing$, *and* $E; P \vdash \Gamma', y : \langle G_1 \big| p \rangle, x : \langle G_2 \big| p \rangle$
- *If* $E; x^{pq}(y).(P \parallel Q) \vdash \Gamma$, *then* $\Gamma \simeq \Gamma_1, \Gamma_2, x : \langle G \big| \{p\} \cup S \rangle$, *with* $G = q \to \tilde{p} : \langle G_1 \rangle; G_2$ *engaged on* $\{p\}$, $S \cap \mathrm{fn}(G) = \varnothing$, $E; P \vdash \Gamma_1, y : \langle G_1 \big| p \rangle$, $E; Q \vdash \Gamma_2, x : \langle G_2 \big| p \rangle$, *and* $p \in \tilde{p}$
- *If* $E; \overline{x}^{p\tilde{q}} \triangleright l.P \vdash \Gamma$, *then* $\Gamma \simeq \Gamma', x : \langle G' \big| \{p\} \cup S \rangle$, *with* $G' = p \to \tilde{q} : \&l; G$ *engaged on* $\{p\}$, $S \cap \mathrm{fn}(G') = \varnothing$, *and* $E; P \vdash \Gamma', x : \langle G \big| p \rangle$
- *If* $E; x^{pq} \triangleleft \{l_1 : P_1, \ldots, l_n : P_n\} \vdash \Gamma$, *then* $\Gamma \simeq \Gamma', x : \langle G \big| \{p\} \cup S \rangle$, *with* $p \in \tilde{p}$, $G = (q \to \tilde{p} : \&l_1; G_1) \& \ldots \& (q \to \tilde{p} : \&l_n; G_n)$ *engaged on* $\{p\}$, $S \cap \mathrm{fn}(G) = \varnothing$, $E; P_1 \vdash \Gamma', x : \langle G_1 \big| p \rangle$, ..., $E; P_n \vdash \Gamma', x : \langle G_n \big| p \rangle$.
- *If* $E; \mathrm{close}(x) \vdash \Gamma$, *then* $\Gamma \simeq \Gamma', x : \langle \mathrm{close} \big| S \rangle$
- *If* $E; \mathrm{wait}(x).P \vdash \Gamma$, *then* $\Gamma \simeq \Gamma', x : \langle \mathrm{end} \big| S \rangle$, $S \neq \varnothing$, *and* $E; P \vdash \Gamma'$
- *If* $E; (P \big|_x Q) \vdash \Gamma$, *then* $\Gamma \simeq \Gamma_1, \Gamma_2, x : \langle G_3 \big| S_1 \uplus S_2 \rangle$, *and* $E; P \vdash \Gamma_1, x : \langle G_1 \big| S_1 \rangle$, $E; Q \vdash \Gamma_2, x : \langle G_2 \big| S_2 \rangle$, G_1 *engaged on* S_1, G_2 *engaged on* S_2, *and* $G_3 \simeq_{S_1 \uplus S_2} G_1 \;{}^{S_1}\!\vee^{S_2}\; G_2$
- *If* $E; (\nu x)P \vdash \Gamma$, *then* $E; P \vdash \Gamma, x : \langle G \big| S \rangle$, *and* $G \downarrow S$
- *If* E; if M then P else $Q \vdash \Gamma$, *then* $E \vdash_D M : \mathrm{Bool}$ *and, for some* Γ_1 *and* Γ_2, *we have* $\Gamma \simeq \Gamma_1 \oplus \Gamma_2$, $E; P \vdash \Gamma_1$, *and* $E; Q \vdash \Gamma_2$

Proof By induction on $E; P \vdash \Gamma$, using Theorem 7.1 to avoid impossible cases. □

Theorem 8.1 *(Subject equivalence) If* $E; P \vdash \Gamma$ *and* $P \equiv Q$ *then* $E; Q \vdash \Gamma$.

Proof By induction on $P \equiv Q$, and using Lemmas 8.1 and 7.3. $\square$

Thanks to this result, from now on, we can consider processes equal modulo $\equiv$.

Lemma 8.2 *If $x \in \mathrm{fn}(P)$ and $E; P \vdash \Gamma$ then for some G, S: $x : \langle G \big| S\rangle \in \Gamma$.*

Proof By induction on $E; P \vdash \Gamma$. $\square$

Lemma 8.3 *If $E; P[x/y, z] \vdash \Gamma, x : \langle G \big| \varnothing\rangle$, then $G \simeq_S$ close, and $E; P \vdash \Gamma, y : \langle \mathrm{close} \big| \varnothing\rangle, z : \langle \mathrm{close} \big| \varnothing\rangle$.*

Proof By induction on $E; P[x/y, z] \vdash \Gamma, x : \langle G \big| \varnothing\rangle$. $\square$

We can now prove the important result of subject reduction.

Theorem 8.2 (subject reduction) *If $E; P_1 \vdash \Gamma_1$ and $P_1 \overset{\alpha}{\to} P_2$, then for some Γ_2, we have $E; P_2 \vdash \Gamma_2$ and $\Gamma_1 \overset{\alpha}{\to} \Gamma_2$.*

Proof By induction on $P_1 \overset{\alpha}{\to} P_2$, using Lemma 8.1.

- $(send)$ We have $E; \overline{x}^{p\tilde{q}}(y).R\big|_x \Pi_i^x (x^{q_i p}(y).(P_i \parallel Q_i)) \vdash \Gamma$, and by Lemma 8.1, we have that $E; \overline{x}^{p\tilde{q}}(y).R \vdash \Gamma', x : \langle p \to \tilde{q} : \langle G\rangle; G' \big| p\rangle$, $E; R \vdash \Gamma', x : \langle p \to \tilde{q} : \langle G\rangle; G' \big| p\rangle$, $E; x^{q_i p}(y).(P_i \parallel Q_i) \vdash \Gamma_i, \Gamma_i', x : \langle p \to q_i : \langle G_i\rangle; G_i' \big| q_i\rangle$, $E; P_i \vdash \Gamma_i, y : \langle G_i \big| q_i\rangle$, $E; Q_i \vdash \Gamma_i', x : \langle G_i' \big| q_i\rangle$, and $\Gamma \simeq \Gamma', \vec{\Gamma}_i, \vec{\Gamma}_i', x : \langle p \to \tilde{q} : \langle G''\rangle; G''' \big| S\rangle$. We can show that p and all the q_i belong to S, and, using Lemma 7.13, we can show that G'' is the merging of G and all the G_i, while G''' is the merging of G' and all of the G_i''. Moreover, $G'' \downarrow S$. Therefore, we have that $E; (\nu y)(R\big|_y \Pi_i^y P_i)\big|_x \Pi_i^x Q_i \vdash \Gamma', \vec{\Gamma}_i, \vec{\Gamma}_i', x : \langle G''' \big| S\rangle$.
- $(case)$ similarly;
- $(comm)$ similarly;
- $(wait)$ easy, using Lemma 8.1;
- $(choice_1)$, easy, using Lemma 8.1;
- $(choice_2)$ Contextual closure by $((\nu x)\_)$: by induction hypothesis, and on case analysis whether the action has the form $x : \gamma$;

  Contextual closure by $(\_\big|_x R)$: by induction hypothesis and on case analysis whether the action has the form $x : \gamma$. In this case, the action is on the session shared with R, and hence using Lemma 7.12 we have to move the action on top of the merged type. Otherwise, the process R is not involved in the communication and we can conclude by induction hypothesis.

$\square$

Remark 8.1 In earlier work about MPST (see, e.g., [29]), subject reduction usually requires some *consistency* condition over the typing environment Γ. In our development, this condition is not explicitly needed because the typing rules for processes ensure that environments are consistent, i.e., the derivability of $E; P_1 \vdash \Gamma_1$ implies that no session in Γ_1 has the type ω.

Progress In usual session types, the progress property means that well-typed systems can always proceed, and in particular they are deadlock-free. In our case, well-typed systems can still contain processes which cannot proceed, not due to a deadlock or miscommunication, but due to some missing participant.

Example 8.1 Let us consider $P = \overline{x}^{pq} \triangleright \&l.\mathrm{close}(x)$. This process is typable $(\varnothing; P \vdash x : \langle p \to q : \&l; \mathrm{close} \big| p\rangle)$, yet it is stuck. It can be completed into a redex $P\big|_x Q$, with $Q = x^{qp} \triangleleft \{l : Q'\}$. In fact, P can be seen as the *restriction* of $P\big|_x Q$ on session x with participants in $\{p\}$. Hence, P is preempted by x and so it can be considered a correct process, waiting for the missing participant.

Therefore, in order to define the progress property for our system, we need to define the restriction of a process to a given set of local participants.

Definition 8.3 (Restriction) We define the restriction of a term P on session x with participants in S (noted $P \downharpoonright_S x$) as follows:

$$
\begin{aligned}
\overline{x}^{p\tilde{q}}\langle M\rangle.P \downharpoonright_S x &= \mathrm{close}(x) \text{ if } p \notin S\\
x^{pq}(y : A).P \downharpoonright_S x &= \mathrm{close}(x) \text{ if } p \notin S\\
\overline{x}^{p\tilde{q}}(y).P \downharpoonright_S x &= \mathrm{close}(x) \text{ if } p \notin S\\
x^{pq}(y).(P \parallel Q) \downharpoonright_S x &= \mathrm{close}(x) \text{ if } p \notin S\\
\overline{x}^{p\tilde{q}} \triangleright l.P \downharpoonright_S x &= \mathrm{close}(x) \text{ if } p \notin S\\
x^{pq} \triangleleft \{l_1 : P_1, \ldots, l_n : P_n\} \downharpoonright_S x &= \mathrm{close}(x) \text{ if } p \notin S\\
P\big|_x Q \downharpoonright_S x &= (P \downharpoonright_S x)\big|_x (Q \downharpoonright_S x)\\
P \downharpoonright_S x &= P \text{ otherwise.}
\end{aligned}
$$

Definition 8.4 (Preemption) We say that a session x with type $G \in \mathfrak{G}$ and local participants S *preempts* P (noted $x : \langle G \big| S\rangle \gg_{\mathrm{g}} P$) when one of these condition occurs:

- $x : \langle p \to \tilde{q} : A; G_2 \big| S\rangle \gg_{\mathrm{g}} ((\overline{x}^{p\tilde{q}}(M).R\big|_x \Pi_i^x (x^{q_i p}(y : A).P_i))) \downharpoonright_S x)\big|_x P$ if $G_2 \simeq_S C$ where C is terminal, or $x : \langle G_2 \big| S - \{p, \tilde{q}\}\rangle \gg_{\mathrm{g}} P$
- $x : \langle p \to \tilde{q} : \langle G_1\rangle; G_2 \big| S\rangle \gg_{\mathrm{g}} ((\overline{x}^{p\tilde{q}}(y).R\big|_x \Pi_i^x (x^{q_i p}(y).(P_i \parallel Q_i))) \downharpoonright_S x)\big|_x P$ if $G_2 \simeq_S C$ where C is terminal, or $x : \langle G_2 \big| S - \{p, \tilde{q}\}\rangle \gg_{\mathrm{g}} P$
- $x : \langle p \to \tilde{q} : \&l; G \big| S\rangle \gg_{\mathrm{g}} (\overline{x}^{p\tilde{q}} \triangleright l.R\big|_x \Pi_j^x x^{q_j p} \triangleleft \{\ldots, l : P_i, \ldots\} \downharpoonright_S x)\big|_x P$ if $G_2 \simeq_S C$ where C is terminal, or $x : \langle G \big| S - \{p, \tilde{q}\}\rangle \gg_{\mathrm{g}} P$

- $x : \langle \text{close} \big| S \rangle \gg_{\text{g}} \text{close}(x)$
- $x : \langle \text{end} \big| S \rangle \gg_{\text{g}} \text{wait}(x).P$
- $x : \langle G_1 \oplus G_2 \big| S \rangle \gg_{\text{g}} P$ if $x : \langle G_1 \big| S \rangle \gg_{\text{g}} P$ or $x : \langle G_2 \big| S \rangle \gg_{\text{g}} P$
- $x : \langle G_1 \,\&\, G_2 \big| S \rangle \gg_{\text{g}} P$ if $x : \langle G_1 \big| S \rangle \gg_{\text{g}} P$ and $x : \langle G_2 \big| S \rangle \gg_{\text{g}} P$
- $x : \langle G \big| S \rangle \gg_{\text{g}} P$ if $x : \langle G \big| S \rangle \gg_{\text{g}} P'$ and $P \equiv P'$

Definition 8.5 (Contextual preemption) We define $x : \langle G \big| S \rangle \gg_{\text{c}} P$ if for some $\mathcal{C}[\,]$, P', we have that $P \equiv \mathcal{C}[P']$, $x \notin \text{fn}(\mathcal{C}[\,])$, and $x : \langle G \big| S \rangle \gg_{\text{g}} P'$.

Intuitively, $x : \langle G \big| S \rangle \gg_{\text{c}} P$ means that every local participant in S is ready to trigger its respective communication described in G. As a consequence, there is no deadlock for x: if all the concerned participants are present there is a redex, otherwise we are blocked due to the absence of some sender or receiver.

Lemma 8.4 *1. If $x : \langle G \big| S_1 \rangle \gg_{\text{g}} P$ and $S_2 \cap \text{fn}(G) = \varnothing$, then $x : \langle G \big| S_1 {\cup} S_2 \rangle \gg_{\text{g}} P$*
2. If $x : \langle G \big| S_1 \rangle \gg_{\text{c}} P$ and $S_2 \cap \text{fn}(G) = \varnothing$, then $x : \langle G \big| S_1 \cup S_2 \rangle \gg_{\text{c}} P$.

Proof 1. By induction on $x : \langle G \big| S_1 \rangle \gg_{\text{g}} P$. 2. Trivial, using point 1. □

Lemma 8.5 *1. if $G_1 \simeq_S G_2$ and $x : \langle G_1 \big| S \rangle \gg_{\text{g}} P$, then $x : \langle G_2 \big| S \rangle \gg_{\text{g}} P$.*
2. if $G_1 \simeq_S G_2$ and $x : \langle G_1 \big| S \rangle \gg_{\text{c}} P$, then $x : \langle G_2 \big| S \rangle \gg_{\text{c}} P$

Proof 1. By induction on $G_1 \simeq_S G_2$. Some cases cannot happen, because $G_1, G_2 \in \mathfrak{G}$, and therefore $G_1 \not\simeq_S \omega$
2. Trivial, using point 1. □

Lemma 8.6 *1. if $x : \langle G_1 \big| S_1 \rangle \gg_{\text{g}} P$, $x : \langle G_2 \big| S_2 \rangle \gg_{\text{g}} Q$ and $G_1 \,{}^{S_1}\!\vee^{S_2}\, G_2 \simeq_{S_1 \uplus S_2} G_3$, then $x : \langle G_3 \big| S_1 \uplus S_2 \rangle \gg_{\text{g}} P \big|_x Q$*
2. if $x : \langle G_1 \big| S_1 \rangle \gg_{\text{c}} P$, $x : \langle G_2 \big| S_2 \rangle \gg_{\text{c}} Q$ and $G_1 \,{}^{S_1}\!\vee^{S_2}\, G_2 \simeq_{S_1 \uplus S_2} G_3$, then $x : \langle G_3 \big| S_1 \uplus S_2 \rangle \gg_{\text{c}} P \big|_x Q$

Proof 1. Using Lemma 8.5, we can reason by induction on a DNF of G_1, G_2.
2. We know that $P \equiv \mathcal{C}_1[P']$ and that $Q \equiv \mathcal{C}_2[Q']$, such that $x \notin \text{fn}(\mathcal{C}_1[\,], \mathcal{C}_2[\,])$, $x : \langle G_1 \big| S_1 \rangle \gg_{\text{g}} P'$ and $x : \langle G_2 \big| S_2 \rangle \gg_{\text{g}} Q'$. We can show by induction on $\mathcal{C}_1[\,]$ and $\mathcal{C}_2[\,]$ that there is some $\mathcal{C}_3[\,]$ such that $P \big|_x Q \equiv C_3[P' \big|_x Q']$, and we can conclude using point 1. □

The following lemma states that if a session is finalized and preempted, then the process (with the session restricted) contains a redex.

Lemma 8.7 *1. If $G \downarrow S$ and $x : \langle G \big| S \rangle \gg_{\text{g}} P$, then $(\nu x)P$ has a redex.*
2. If $G \downarrow S$ and $x : \langle G \big| S \rangle \gg_{\text{c}} P$, then $(\nu x)P$ has a redex.

Proof 1. By induction on $x : \langle G \big| S \rangle \gg_{\text{g}} P$, using the hypothesis $G \downarrow S$ to show we have a full redex.
2. We have that $P \equiv \mathcal{C}[P']$ where $x : \langle G \big| S \rangle \gg_{\text{g}} P'$ and $x \notin \text{fn}(\mathcal{C}[\,])$. We can show that $(\nu x)P \equiv \mathcal{C}[(\nu x)P']$, and we can conclude using point 1. □

We now can prove progress.

Theorem 8.3 *(Progress) If $E; P \vdash \Gamma$, then there is a redex in P, or for some $x : \langle G \big| S \rangle \in \Gamma$ we have $x : \langle G \big| S \rangle \gg_{\text{c}} P$.*

Proof The proof is done by induction on the typing derivation $E; P \vdash \Gamma$.

- Rules $(sendvalue)$, $(recvvalue)$, $(send)$, $(recv)$, (sel_i), $(case)$, $(close)$, $(wait)$: we have immediately the preemption of a channel.
- Rules $(weaken)$, $(contract)$: we can easily apply the induction hypothesis.
- Rule $(extra)$: by Lemma 8.4.
- Rule $(\simeq)$: by Lemma 8.5.
- Rule $(\big|)$: in this case the last rule of the typing derivation is as follows:

$$\frac{E; P \vdash \Gamma_1, x : \langle G_1 \big| S_1 \rangle \quad E; Q \vdash \Gamma_2, x : \langle G_2 \big| S_2 \rangle \quad G_3 \simeq_{S_1 \uplus S_2} G_1 \,{}^{S_1}\!\vee^{S_2}\, G_2}{E; P \big|_x Q \vdash \Gamma_1, \Gamma_2, x : \langle G_3 \big| S_1 \uplus S_2 \rangle}$$

  If $x : \langle G_1 \big| S_1 \rangle \gg_{\text{c}} P$ and $x : \langle G_2 \big| S_2 \rangle \gg_{\text{c}} Q$, then, using Lemma 8.6, we have that $x : \langle G_3 \big| S_1 \uplus S_2 \rangle \gg_{\text{c}} P \big|_x Q$. Otherwise, we conclude by induction.
- Rule (ν): let's consider

$$\frac{E; P \vdash \Gamma, x : \langle G \big| S \rangle \quad G \downarrow S}{E; (\nu x)P \vdash \Gamma}(\nu)$$

  If $x : \langle G \big| S \rangle \gg_{\text{g}} P$, then, using Lemma 8.7, we have that that $(\nu x)P$ has a redex. Otherwise, we conclude by induction.
- Rule $(\oplus)$: we immediately have a redex. □

Example 8.2 Continuing Example 6.3, we can see that there is no redex in $P_p \big| P_q$, but there is a preemption:

$$x : \langle r \to p : \text{int}; p \to q : \text{int}; q \to r : \text{int};\ \text{close} \big| p, q \rangle \gg_{\text{c}} P_p \big| P_q$$

That is, the first action that $P_p \big| P_q$ is ready to do on session x is $x : r \to p : \text{int}$, but this is not possible because participant r for session x is not present. We can see that P_r is also preempted, but in an incompatible way :

$$x : \langle q \to r : \text{int}; r \to p : \text{int}; \text{close} \big| r \rangle \gg_{\text{c}} P_r$$

That is, the first action that P_r is ready to do is $x : q \to r : \text{int}$, and this means that $P_p \big| P_q \big| P_r$ cannot progress, and in fact it is not typable.

Example 8.3 Continuing Example 6.4, we have that $P_q \big|_x P_r$ is preempted:

$$x : \langle G'_1 \ \&\ G_2 \ \&\ G_3 \big| q, r\rangle \gg_{\text{c}} P_q \big|_x P_r$$

We recall the definition of G'_1, G_2, and G_3:

$$G'_1 := p \to qr : \langle \text{close}\rangle;\ p \to q : \langle \text{close}\rangle; \text{close}$$
$$G_2 := p \to q : \langle \text{close}\rangle;\ p \to qr : \langle \text{close}\rangle; \text{close}$$
$$G_3 := p \to q : \langle \text{close}\rangle;\ p \to q : \langle \text{close}\rangle;\ p \to r : \langle \text{close}\rangle;\ \text{close}$$

The first communication given by G'_1 is $p \to qr : \langle\text{close}\rangle$, the first communication given by G_2 is $p \to q : \langle\text{close}\rangle$, and the first communication given by G_3 is either $p \to q : \langle\text{close}\rangle$ or $p \to r : \langle\text{close}\rangle$, because both of these communications are independent relatively to $\{q, r\}$. As a consequence, the preemption of $P_q \big|_x P_r$ means that it is ready to the action $x : p \to qr : \langle\text{close}\rangle$, and it is also ready to do the action $x : p \to q : \langle\text{close}\rangle$, and also the action $x : p \to r : \langle\text{close}\rangle$. The action that will really be done is chosen externally. Of course, it does not mean that these three actions will be executed; in this example, participant r only expects one message from p, so if the action $x : p \to qr : \langle\text{close}\rangle$ is executed, then $x : p \to r : \langle\text{close}\rangle$ cannot be executed afterwards.

9 Conclusions

In this paper, we have introduced *partial sessions* and *partial (multiparty) session types*, extending global session types with the possibility to type also *open* systems, i.e., sessions with missing participants. Sessions with the same name but observed by different participants can be merged if their types are *compatible*; in this case, the type for the unified session can be derived compositionally from the types of components. To this end, we have provided a merging algorithm, which allows us to detect incompatible types, due to miscommunications or deadlocks, as early as possible; this differs from usual session type systems which delay all the checks to when the system is completed (i.e., at the restriction rule). Therefore, in this theory the distinction between local and global types vanishes: local types correspond to partial session types for sessions with a single participant, and global types correspond to *finalized* partial session types, i.e., in which no participant is missing. We have also generalized the notion of *progress* to accommodate the case when a partial session cannot progress not due to a deadlock, but to some missing participant.

Future work. An interesting application of partial session types would be in the verification of composition of components, like, e.g., containers *à la* Docker; to this end, we can think of defining a typing discipline similar to the one presented in this paper, but tailored for a formal models of containers, like that in [8].

We conjecture that, for the type system presented in this paper, both type checking and type inference are decidable. The idea is that, in order to be typable, the structure of a process has to match the structure of the type(s), up-to type equivalence; hence, the typing derivation is bounded by the complexity of process terms. At worst, this bound is exponential, as in the application of type equivalence rule we have to explore a possibly exponential space of equivalent types; however, this limit could be improved by some algorithmic machinery concerning the normal form of types, which we leave to future work.

The current merging algorithm returns types that may contain many equivalent subterms; a future work could be to define shorter and more efficient representations. Another interesting aspect of this algorithm is that it is defined by two functions (map and mcomm), which can be updated separately in future variations; in particular, adding recursion only requires to update the function map, while adding new kinds of communication, or changing how communications are merged, only requires to update the function mcomm.

In this paper, we have considered a calculus with synchronous multicast, along the lines of [11,32] and others. However, it would be interesting to extend the definitions and results of this paper to an *asynchronous* version of the calculus, or a calculus where sessions can merge at runtime, akin the names in the Fusion calculus [25]. This is not immediate, as it requires non-trivial changes in the typing systems and especially in the (already quite complex) merging operation.

Following the Liskov substitution principle, we could define a subtyping relation by seeing & and ⊕ as the meet and join operator of a lattice, respectively. However, a semantical understanding of this subtyping relation is not clear yet.

One intriguing possible extension would be to add some form of *encapsulation*. For instance, if we have the type $p \to q : m_1; q \to r : m_2; p \to r : m_3;$ close from the viewpoint of $\{q, r\}$, then we could be tempted to "erase" the communication $q \to r : m_2$, since this communication is purely internal, but this erasure would not be compatible with equivalence:

$$p \to q : m_1;\ q \to r : m_2;\ p \to r : m_3;\ \text{close} \not\simeq_{\{q,r\}}$$
$$p \to q : m_3;\ q \to r : m_2;$$
$$p \to r : m_1;\ \text{close}$$

but $p \rightarrow q : m_1; p \rightarrow r : m_3$; close $\simeq_{\{q,r\}} p \rightarrow q : m_3; p \rightarrow r : m_1$; close. How to add a form of encapsulation to our type system is an open question.

Finally, to guarantee the correctness of most complex proofs and definitions of this paper, it would be useful to formalize them in a proof assistant, like Coq.

Acknowledgements This work has been supported by the Italian MIUR project PRIN 2017FTXR7S *IT MATTERS* (Methods and Tools for Trustworthy Smart Systems). The authors are grateful to Mariangiola Dezani-Ciancaglini, Marco Peressotti and the anonymous reviewers for their useful remarks on the preliminar version of this paper.

Funding Open access funding provided by Università degli Studi di Udine within the CRUI-CARE Agreement.

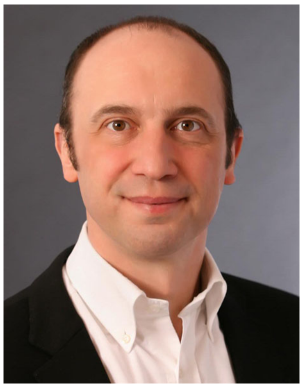
References

1. Atkey, R.: Observed communication semantics for classical processes. In: Yang, H. (ed.) Programming Languages and Systems, pp. 56–82. Springer, Berlin (2017)
2. Barbanera, F., de' Liguoro, U., Hennicker, R.: Global types for open systems. In: Proceedings of the ICE, volume 279 of EPTCS, pp. 4–20 (2018)
3. Barbanera, F., Dezani-Ciancaglini, M.: Open multiparty sessions. In: Proceedings of the ICE, volume 304 of EPTCS, pp. 77–96 (2019)
4. Barbanera, F., Dezani-Ciancaglini, M., Lanese, I., Tuosto, E.: Composition and decomposition of multiparty sessions. J. Log. Algebraic Methods Program. **119**, 100620 (2021)
5. Barbanera, F., de'Liguoro, U., Hennicker, R.: Connecting open systems of communicating finite state machines. J. Log. Algebraic Methods Programm. **109**, 100476 (2019)
6. Barbanera, F., Lanese, I., Tuosto, E.: Composing communicating systems, synchronously. In: ISoLA, volume 12476 of Lecture Notes in Computer Science, pp. 39–59. Springer (2020)
7. Barr, M.: *-autonomous categories and linear logic. Math. Struct. Comput. Sci. **1**(2), 159–178 (1991)
8. Burco, F., Miculan, M., Peressotti, M.: Towards a formal model for composable container systems. In: Hung, C., Cerný, T., Shin, D., Bechini, A. (eds.), SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, pp. 173–175. ACM (2020)
9. Caires, L., Vieira, H.T.: Conversation types. Theor. Comput. Sci. **411**(51–52), 4399–4440 (2010)
10. Carbone, M., Montesi, F., Schürmann, C.: Choreographies, logically. Distrib. Comput. **31**(1), 51–67 (2018)
11. Carbone, M., Montesi, F., Schürmann, C., Yoshida, N.: Multiparty session types as coherence proofs. Acta Informatica **54**(3), 243–269 (2017)
12. Castellani, I., Dezani-Ciancaglini, M., Giannini, P.: Event structure semantics for multiparty sessions. In: Models, Languages, and Tools for Concurrent and Distributed Programming, pp. 340–363. Springer, (2019)
13. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L.: Global progress for dynamically interleaved multiparty sessions. Math. Struct. Comput. Sci. **26**(2), 238–302 (2016)
14. De Nicola, R., Melgratti, H.: Multiparty testing preorders. In: Trustworthy Global Computing, pp. 16–31. Springer, (2015)
15. Girard, J.-Y.: Linear logic. Theor. Comput. Sci. **50**(1), 1–101 (1987)
16. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.), Proceedings of the ESOP'98, volume 1381 of Lecture Notes in Computer Science, pp. 122–138. Springer, (1998)
17. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Necula, G., Wadler, P. (eds.), POPL 2008, pp. 273–284. ACM, (2008)
18. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9:1-9:67 (2016)
19. Jespersen, T.B.L., Munksgaard, P., Larsen, K.F.: Session types for Rust. In: Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, pp. 13–22 (2015)
20. Keizer, A.C., Basold, H., Pérez, J.A.: Session coalgebras: a coalgebraic view on session types and communication protocols. In: Yoshida, N. (ed.), Proceedings of the ESOP 2021, Proceedings, volume 12648 of Lecture Notes in Computer Science, pp. 375–403. Springer, (2021)
21. Kokke, W., Montesi, F., Peressotti, M.: Better late than never: a fully-abstract semantics for classical processes. In: Proceedings of the ACM on Programming Languages, 3(POPL), (2019)
22. Lange, J., Tuosto, E.: Synthesising choreographies from local session types. In: CONCUR, volume 7454 of Lecture Notes in Computer Science, pp. 225–239. Springer, (2012)
23. Lange, J., Tuosto, E., Yoshida, N.: From communicating machines to graphical choreographies. In: POPL, pp. 221–232. ACM, (2015)
24. Merkel, D.: Docker: lightweight Linux containers for consistent development and deployment. Linux J. **2014**(239), 2 (2014)
25. Miculan, M.: A categorical model of the Fusion calculus. In: Proceedings of the MFPS, volume 218 of Electronic Notes in Theoretical Computer Science, pp. 275–293. Elsevier, (2008)
26. Montesi, F., Yoshida, N.: Compositional choreographies. In: Proceedings of the CONCUR, volume 8052 of Lecture Notes in Computer Science, pp. 425–439. Springer, (2013)
27. Neubauer, M., Thiemann, P.: An implementation of session types. In: International Symposium on Practical Aspects of Declarative Languages, pp. 56–70. Springer, (2004)
28. Nielsen, M., Winskel, G.: Models for concurrency. In: Proceedings of the Mathematical Foundations of Computer Science, pp. 43–46 (1991)
29. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. In: Proceedings of the ACM on Programming Languages, vol. 3(POPL), pp. 1–29, (2019)
30. Stolze, C., Miculan, M., Di Gianantonio, P.: Composable partial multiparty session types. In: Salaün, G., Wijs, A. (eds.), Formal Aspects of Component Software—17th International Conference, FACS 2021, Proceedings, volume 13077 of Lecture Notes in Computer Science, pp. 44–62. Springer, (2021)
31. Toninho, B., Yoshida, N.: Polymorphic session processes as morphisms. In: Alvim, M.S., Chatzikokolakis, K., Olarte, C., Valencia, F. (eds.), The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy—Essays Dedicated to Catuscia Palamidessi on the Occasion of Her 60th

Birthday, volume 11760 of Lecture Notes in Computer Science, pp. 101–117. Springer, (2019)

32. Wadler, P.: Propositions as sessions. J. Funct. Programm. **24**(2–3), 384–418 (2014)

Claude Stolze received a Ph.D. in Computer Science at INRIA Sophia-Antipolis (France) in 2019. He is currently a post-doc at the university of Udine, where he studies session types. His research mainly concerns type theory, including intersection, union, and dependent types.



Marino Miculan received the Ph.D. in Computer Science from the University of Pisa (Italy) in 1997. Since 2005 he is Associate Professor at the University of Udine (Italy), where he directs the Laboratory of Models and Application of Distributed System and the Udine node of the CINI National Laboratory on Cybersecurity. His research mainly concerns semantic models and formal methods for concurrent and distributed systems, in particular for ensuring security properties. He is (co)author of more than 75 publications in international scientific journals and conference proceedings with peer review. He has served as member or chair to the program committees of several international conferences and workshops.

Pietro Di Gianantonio received his PhD in Computer Science from the University of Pisa (Italy) in 1993. He has been a visiting researcher at Imperial College, University of Edinburgh, CWI Amsterdam. Since 2001, he is an Associate Professor at the University of Udine (Italy). His interests include computable analysis, type assignment systems, and semantics of programming languages.